Technical Report 988

TYPICAL: A Knowledge Representation System for Automated Discovery and Inference

Kenneth W. Haase Jr.

MIT Artificial Intelligence Laboratory

This blank page was inserted to preserve pagination.

TYPICAL
A Knowledge Representation System

based on

Type Specification

for

Automated Discovery and Inference

Kenneth W. Haase Jr.

TYPICAL is a package for describing and making automatic inferences about a broad class of SCHEME predicate functions. These functions, called *types* following popular usage, delineate classes of primitive SCHEME objects, composite data structures, and abstract descriptions. TYPICAL types are generated by an extensible combinator language from either existing types or primitive terminals. These generated types are located in a lattice of predicate subsumption which captures neccessary entailment between types; if satisfaction of one type neccessarily entails satisfaction of another, the first type is below the second in the lattice. The inferences made by TYPICAL are relations in this lattice of subsumption; when a type is defined, TYPICAL computes the position of the new definition within the lattice and establishes it there. This information is then accessible to both later inferences and other programs (reasoning systems, code analyzers, etc) which may need the information for their own purposes. TYPICAL was developed as a representation language for the discovery program Cyrano; particular examples are given of TYPICAL's application in the Cyrano program.

This is a considerably revised version of a thesis submitted to the Department of Electrical Engineering and Computer Science on May 9, 1986 in partial fulfillment of the degree of Master of Science.

This report describes research done in part at the Artificial Intelligence Laboratory of the Massachussets Institute of Technology. Much of the support for the laboratory's research is provided by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-85-K-0124. Of course, the opinions in this thesis are those of the author and in no way reflect the opinions of the Department of Defense or the US Government.

This empty page was substituted for a blank page in the original document.

Acknowledgements

The work described in this thesis, and the larger work surrounding it, has been supported by many people. Intellectually, it owes a debt to the overall environment of the Artificial Intelligence Laboratory at MIT; personally, the author owes a great debt to people who have kept him stimulated and sane in the time this work was progressing.

First and foremost, Marvin Minsky has been a source of inspiration and support throughout my career at MIT. This thesis is a 'practical thesis' describing how a particular program works, rather than offering a theory of mind or mental activity. However, the attitudes and perspectives which endgendered this implementation arise from a way of looking at the world and the mind which is Marvin's gift and lesson to all his students.

The 7th and 4th floor communities at the AI lab has been a source and sounding board for a multitude of perspectives and ideas; in particular, David McAllester, David Chapman, Jonathan Rees, and John Mallery have often listened and provided valuable comments. David McAllester found some awful bugs and told me about them; I'm even more grateful than I am disappointed (which is saying a lot).

The Scheme group at MIT has been a source of superlative technical and intellectual support. In addition, a generous equipment grant from Hewlett-Packard Corporation allowed this thesis to be completed in the seclusion of the author's office.

Patrick Winston helped me stay below the clouds and in my right mind. He has been a filter for many of my crazier ideas and an amplifier for a few. David Kirsh has given me perspective in numerous conversations and discussions; his explanations back to me ('Now let's see if I have this correctly.....') have made many of my own ideas much the clearer. Ramesh Patil shared a genuine interest the problems I thought were important and provided inspired insights into exposition.

My housemates kept a warm home which contributed greatly to my sanity; thank you Ed, David, Hazel, Jan, Bruce and Paul. Other friends helped with maintaining the remainder of my sanity. Thank you Gordon, Mary, Lisa, Nat, Bunny, Polly, and Sandy. I love you all.

Beacon Hill Friends Meeting and the Haley House community have been a source of peace and growth over the past years. It is difficult to imagine life without them; fortunately I don't have to. God bless all of you.

Much of the support for this research was provided by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-85-K-0124. Of course, the opinions in this thesis are those of the author and in no way reflect the opinions of the Department of Defense or the US Government. In turn, the opinions of the Department of Defense also in no way reflect the opinions of the author.

Finally, my family has supported me with care and love for 25 years so far, and show no signs of letting up. Thank you for all of your love and support. My love will always be with you.

This empty page was substituted for a blank page in the original document.

For my brother,

This empty page was substituted for a blank page in the original document.

CONTENTS

Appendices

This empty page was substituted for a blank page in the original document.

This empty page was substituted for a blank page in the original document.

Chapter 1
TYPICAL Introduction

TYPICAL is a package which makes automatic inferences about a broad class of SCHEME predicate functions. These functions, called *types* following popular usage, delineate potentially overlapping classes of primitive SCHEME objects and composite data structures. TYPICAL types are generated by an extensible combinator language from either existing types or particular sorts of SCHEME objects. Generated types are located in a lattice of predicate subsumption which represents neccessary entailment between types; if one type is below another in the lattice, satisfaction of the first type neccessarily entails satisfaction of the second. The inferences made by TYPICAL are relations in this lattice of subsumption; when a type is defined, TYPICAL computes the position of the new definition within the lattice and establishes it there. This information is then accessible to both later inferences and other programs (reasoning systems, code analyzers, etc) which may need the information for their own purposes.

TYPICAL was developed as representational support for **Cyrano**, an automated discovery program which proposes and analyzes concepts and conjectures in elementary mathematics and several other domains. The principle behind **Cyrano**'s design is a view of discovery as a process of extending a conceptual vocabulary; a discovery program is given a conceptual vocabulary and, based on its observed empirical properties, produces an extended or modified vocabulary which is then the basis for further empirical analysis and extension. For a discovery process to be effective, newly developed concepts must be represented in a structurally similar form to initial starting concepts. To satisfy this constraint, **Cyrano**'s initial and evolved conceptual vocabularies are described uniformly in TYPICAL; new concepts and definitions are constructed by TYPICAL combinators and placed in the lattice of types. **Cyrano**'s inferences about the neccessary properties of its definitions (the

type combinations it generates) are all handled by TYPICAL; likely or heuristic inferences are handled by **Cyrano**.

While developed as representational support for **Cyrano**, TYPICAL soon found a place in the program's control structure; the power of an organized lattice of predicates became useful for specifying tests and conditions in the program. This trend was further encouraged by an interest in making **Cyrano** able to reflect on its own control structure; having the representation and control structure share syntax and structure was a way of supporting this. In particular, the trigger conditions of heuristics and the experimental confirmation of empirical regularities are both described in the lattice. Chapters 5 and 6 present examples of TYPICAL's application in the **Cyrano** program.

1.1 Technical Contributions

This thesis makes several technical contributions in TYPICAL's own design and implementation:

- The combinator language allowing the definition of new types and new combinators which fit within the existing framework of the lattice.
- Subsumption principles for a mixed vocabulary of combinators including definitions like power sets, functional 'role' constraints, and simple recursive definitions.
- An implementation of uncertainty which allows types to be partially specified; predicates in the lattice may return either 'true,' 'false,' or 'i don't know'.
- Efficient and sound algorithms for placing conjunctions, disjunctions, and other predicate combinations in the lattice.

In addition, Chapters 5 and 6 present the use of TYPICAL in the discovery program **Cyrano**. These illustrate TYPICAL's application to more traditional 'AI-style' problems:

- Chapter 5 presents the *indexer* used by **Cyrano** as a taxonomic classifier and heuristic rule engine. Indexing is the backbone of **Cyrano**'s control structure and uses the lattice to represent types of events and situations to which **Cyrano** responds.
- Chapter 6 presents the use of TYPICAL in setting up the 'experiments' by which **Cyrano** confirms or disconfirms empirical properties of its definitions. This uses the indexing facility to notice counterexamples or examples to proposed regularities.

1.2 Related Work

TYPICAL first emerged as a solution to an AI representation problem; it's initial impetus and inspiration came from the tradition of AI languages beginning with FRL [Gol76], moving to KRL [BW77] and UNITS [Ste79], and culminating (some might say) in languages like KL-ONE [BS85] and RLL-1 [Gre80]. Work on Cyrano began in the representation language language ARLO [Haa86], an approach suggested by Lenat's use of RLL-1 in the

discovery program Eurisko[Len83]. Unfortunately, for reasons outlined in the conclusions to [Haa86] and explored in more detail in [Haa87], the ad-hoc extensibility of ARLO (and of frame-based RLL's in general) failed to integrate well with a 'first principles' discovery program.

As a language for specifying data types (as opposed to a general AI representation language), TYPICAL bears surface similarity to the typing systems of languages like Algol-60 [Nau61], Pascal [Wir71], and CLU [LAB*79]. It bears a deeper similarity to the type inference facilities provided by languages like ML [Car83] [Mil83]; however ML's typing of procedures (using Milner's theory of type polymorphism [Mil78]) is not provided by TYPICAL since no provision for type variables and relational types is incorporated into the language. Beyond this, there is a rich literature on typing systems and type inference, but the aim of TYPICAL was — in most respects — orthogonal to the aims of such efforts.

Closer to TYPICAL is the mathematical representation language ONTIC [McA87] which provides a language for describing a range of mathematical concepts to sophisticated proof checker. TYPICAL, however, was designed as a limited inference component for an empirical discovery system and its design requirements were somewhat different.

1.3 Structure of the Thesis

I begin this thesis (Chapter 2) by giving simple examples of TYPICAL in use and the inferences it makes; these are brief snippets which foreshadow the more extensive capabilities deployed in later chapters. This brief exposition is not written for detailed reading; it surveys what TYPICAL does and can be safely skimmed.

After sketching what TYPICAL does, the implementation of TYPICAL is described in detail in Chapter 3: data structures, combinators, the representation of uncertainty, and various efficiency measures are presented.

The details of individual combinators — how they do their inferences — are covered in Chapter 4. All of TYPICAL's primitive combinators are described in this chapter. A formal analysis of these combinators and their inferences is presented in Appendix A-1.

Chapter 5 describes the 'indexer,' a rule engine built around TYPICAL's lattice of types. The indexer combines a taxonomic identifier with the attachment of procedures to types in the lattice. Indexing an object locates it in the lattice of defined types and executes the procedures — classification daemons — attached to the types it satisfies.

Chapter 6 describes how the **Cyrano** program uses TYPICAL to confirm or disconfirm empirical properties. All empirical properties recognizable by Cyrano are translated into types which specify experimental sets of examples and counterexamples. Classification daemons attached to these respective sets catch convincing amounts of evidence or counterevidence to support or discount the program's empirical suspicions.

The appendices begin with a formal analysis of TYPICAL's combinators (Appendix A-1) followed by a brief manual for the use of TYPICAL (Appendix A-2). Finally, Appendix A-3 discusses ways to get copies of TYPICAL for use or experimentation.

This empty page was substituted for a blank page in the original document.

Chapter 2
TYPICAL Use

TYPICAL is a set of SCHEME procedures and data structures. A TYPICAL type is a data structure — a 'type description' — describing a SCHEME predicate and the relation of that predicate to other described predicates. Type descriptions are first class objects; they may be bound to variables, passed as arguments, or subject to typing themselves. New types are defined by a variety of *combinator* procedures; TYPICAL combinators are SCHEME procedures which take either existing type descriptions or particular SCHEME objects and produce type descriptions as results.

New TYPICAL types are positioned in a lattice of predicate subsumption; if a type T is below a type P in the lattice of subsumption (T is *subsumed by* P), it means that satisfaction of T entails satisfaction of P; interpreted as predicates, if T is below P, $T(x) \longrightarrow P(x)$. Informally, you can think of subsumption as being a subset relation between sets of objects. The top of the lattice is everything that is (or might be) in the world; all of the types beneath it are subsets of that set. There are a few problems with this: you don't really have a handle on the set of all possible things in the universe, you can't reel members off one after another, and it violates the foundation axiom of set theory (you can't define a set by a condition on the universe) which leaves open a variety of 'set of all sets' paradoxes. But it is sometimes useful to think about subsumption of types as containment of sets, particularly near the bottom of the lattice where sets are finite and subsumption and containment really are the same.

With regard to the formal mathematical conception of a lattice, there are some differences and various holes. In particular, TYPICAL's lattice implementation has no distinguished BOTTOM element ("$\perp$"). In fact, the bottom fringe of the lattice is cut off in an odd way. As the lattice is descended, subsumption relations connect each type to its specializations; but at the end of the chain of subsumptions — just above where some element

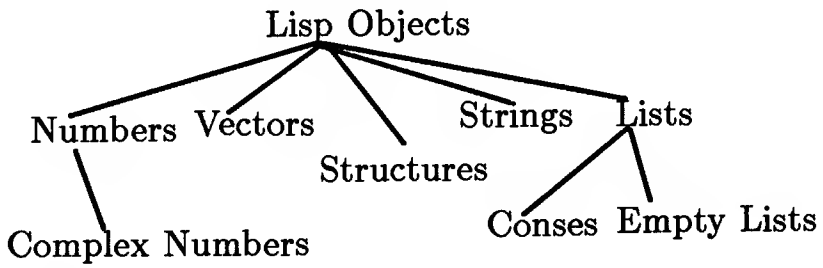

Figure 2-1. The implementation environment of TYPICAL is represented in the lattice by primitive types of descriptions: numbers, lists, symbols, strings, etc.

would touch bottom — the subsumption relation becomes an instantiation or satisfaction relation 'connecting' to the concrete universe of SCHEME objects. The type which would represent the singleton set containing a particular unique object becomes simply the object itself with a satisfaction pointer to the types it satisfies. One might say that 'bottom' is then the unrepresented (and empty) specialization of these individual instances.

Taking the terminology of AI representation langauges, we call the types above a given type its generalizations; the types below a given type are its specializations. In practice, the stored generalizations and specializations of types constitute a minimal generator of the type lattice; P never stores both G and H as generalizations or specializations if G and H are directly related in the lattice themselves. In the case of generalizations, it stores the most specialized; in the case of specializations it stores the most general.

Every type also has — for use by TYPICAL and its applications — a set of arbitrary properties which can be accessed by user functions. These are used for both keeping track of extra-lattice properties and for storing other relations between concepts — like creation relations — which application programs may wish to maintain and refer to.

2.1 Example: SCHEME's Data Types

This section describes how TYPICAL represents its 'implementation environment'. Figure 2-1 shows the lattice of primitive SCHEME data types as represented for TYPICAL. Each of these is defined by building a primitive type description around a provided scheme predicate.

New primitive types are constructed by the `SIMPLE-TYPE` procedure. This procedure takes two arguments: a LISP predicate and a generalization (which is an existing type in the lattice). The resulting type description has the LISP predicate as its determining function and a single generalization which is the generalization given as an argument. For

instance, here we define a few simple SCHEME data types as TYPICAL types:

```
;;; Define the class of LISP objects
(define lisp-objects (simple-type lisp-object? lattice-top))
;;; Numbers are a sort of LISP object,
(define numbers (simple-type number? lisp-objects))
;;; While complex-numbers are a sort of number.
(define complex-numbers (simple-type complex? numbers))
;;; Defining lists, conses, the empty list, vectors, structures, and strings.
(define lists (simple-type list? lisp-objects))
(define conses (simple-type pair? lists))
(define empty-lists (simple-type null? lists))
(define vectors (simple-type vector? lisp-objects))
(define structures (simple-type structure? lisp-objects)
(define strings (simple-type string? lisp-objects))
```

The class LISP-OBJECTS [1] is a class whose predicate accepts anything; all LISP data types (and thus all types of encoded descriptions) are specializations of this. For reasons of epistemic consistency, LISP-OBJECTS is not quite the same as the top of the lattice; there is a proviso for a class of 'protected descriptions' which are not classed as LISP objects but rather as 'things in the world.' This distinction is supposedly captured by the predicate `LISP-OBJECT?`. While not used yet, this distinction may eventually be neccessary for some particular sticky representation problems; no arguments are made, however, for its sufficiency.

The TYPICAL predicate `SUBSUMED-BY?` determines subsumption relations in the lattice; `SUBSUMED-BY?` takes two types as arguments and returns true if the first type is beneath the second in the lattice:

```
(subsumed-by? conses lists) ⟹ #!TRUE
(subsumed-by? lists conses) ⟹ #!FALSE
(subsumed-by? strings lists) ⟹ #!FALSE
```

Subsumption relations between types are immutable; the contract of the lattice demands that any type definition (addition to the lattice) neither create nor destroy existing subsumption relations between types in the lattice.

TYPICAL does not completely represent type complementation or disjointness for complexity reasons (see Section A-1.3, Page 79); however, TYPICAL does represent a limited sort of disjointness which nonetheless has great utility for detecting conflicts and contra-

---

1 In interaction with TYPICAL, types actually print out something like this:
`#[144:#[112:#[20:Integers] X #[20:Integers]]<and>#[131:LESS-THAN-PAIRS]]`
which is how the type of numerically ordered integer pairs might appear. For purposes of explanation, types in this and following chapters will be shown as specially printed tokens (like Numbers or Ordered Integer Pairs) whose names correspond to the identifiers bound to the type.

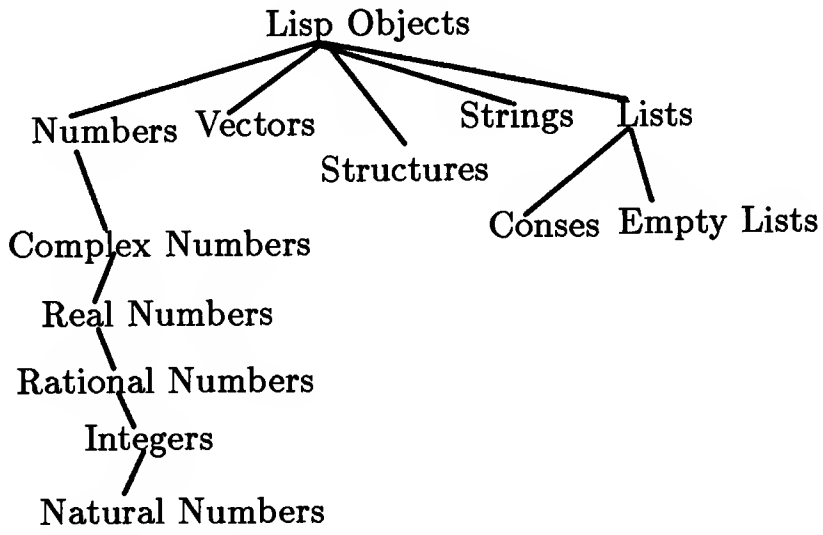

Figure 2-2. The lattice of primitive types can be extended to include SCHEME's taxonomy of abstract numbers.

dictions in representations. Types are declared disjoint by the procedure **MAKE-DISJOINT!** which takes a list of types and declares them all mutually disjoint:

```
;;; Lists and strings and numbers should all be mutually disjoint.
(make-disjoint! lists vectors strings numbers)
;;; And empty lists and conses are disjoint specializations of Lists.
(make-disjoint! conses empty-lists)
```

TYPICAL can make certain inferences about disjointness based on the propogation of disjointness down the lattice; if two types are disjoint, all of their specializations are disjoint. In particular, if we use the predicate **DISJOINT?** to ask about the relation of complex numbers and lists (we defined integers above) and complex numbers and numbers:

```
(disjoint? complex-numbers lists) ⟹ #!TRUE
(disjoint? complex-numbers integers) ⟹ #!FALSE
```

Declarations of disjointness allows some procedures to catch particular errors; for instance, an attempt to intersect two disjoint types signals an error.

Since types are data structures in SCHEME, we can define the meta-type Types beneath Structures. This type, which serves as the root of the sublattice of meta-types, is a **simple-type** with the determining predicate **TYPE-DESCRIPTION?**; since types are implemented as structures, the generalization given Types is the primitive type Structures which we defined above:

```
(define types (simple type type-description? structures))
```

The lattice constructed so far by these definitions has been relatively flat; we can add some depth by extending the representation of numbers to the entire tower of SCHEME

numbers:[2]

```
;;; These are the number data types of SCHEME, which it shares
;;;  with Common LISP and many other programming languages.
(define real-numbers (simple-type real? complex-numbers))
(define rationals (simple-type rational? real-numbers))
(define integers (simple-type integers rationals))
;;; SCHEME makes an additional distinction between EXACT and
;;;  INEXACT numbers; TYPICAL represents these as well:
(define exact-numbers (simple-type exact? numbers))
(define inexact-numbers (simple-type inexact? numbers))
(make-disjoint! exact-numbers inexact-numbers)
```

The resulting (extended) type lattice is depicted in Figure 2-2.

The tower constructed here can, as above, be examined by the predicate SUBSUMED-BY? or its alias <<?:

```
(SUBSUMED-BY? RATIONALS COMPLEX-NUMBERS) ⟹ #!TRUE
(SUBSUMED-BY? COMPLEX-NUMBERS RATIONALS) ⟹ #!FALSE
(<<? TYPES VECTORS)                      ⟹ #!TRUE
(<<? TYPES LISTS)                        ⟹ #!FALSE
```

It is also possible to access the lattice connections of a type directly; the explicit generalizations and specializations can be accessed by SCHEME procedures which return lists of types. The SCHEME procedures GENERALIZATIONS and SPECIALIZATIONS each take a type description as an argument and return a list of the immediate generalizations or specializations of the type. For instance:

```
(GENERALIZATIONS INTEGERS) ⟹ (Rationals)
(SPECIALIZATIONS VECTORS)  ⟹ (Types)
(SPECIALIZATIONS NUMBERS)
  ⟹ (Complex Numbers  Exact Numbers  Inexact Numbers)
```

We can define new types in terms of existing types by using more combinators more complicated than SIMPLE-TYPE. In particular, the <AND> and <OR> combinators intersect or union an arbitrary number of other types. For example, we define the type of exact reals or the type of reals that are either exact or inexact:

```
(define exact-reals (<AND> exact-numbers real-numbers))
(define exact-or-inexact-reals
  (<OR> exact-reals (<AND> inexact-numbers real-numbers)))
```

If exact and inexact numbers were complements, we would like the definition of Exact or Inexact Integers to become the same as Integers; but TYPICAL does not completely represent complementation, so these definitions would remain distinct. In any

[2]The types in this tower do not specify implementation types, but instead implement the abstract numerical data types of [RC86] SCHEME; rather than distinguishing representations as implemented, they make finer and finer distinctions among abstract numbers.

case, since we are only representing the disjointness of exact and inexact integers, the separation in this particular case remains valid: there might be another type — say [Exact on Friday Afternoons] — which actually lay between exact and inexact numbers and thus a type [Reals Exact On Friday Afternoons] between [Exact Reals] and [Inexact Reals].

Inferences about type combinations are emedded in the subsumption lattice; any type subsumes another if the representation of subsumption relations says so. In particular, the information is stored in the generalizations, so that asking for the generalizations or specializations of combined types will simply recall the combination:

```
(GENERALIZATIONS EXACT-INTEGERS) ⟹ ([Exact Numbers] [Integers])
(SPECIALIZATIONS EXACT-OR-INEXACT-INTEGERS)
   ⟹ ([Exact Integers] [Inexact Integers])
```

In many cases though, especially when types being combined are themselves combinations of types, more sophisticated inferences must be made. In TYPICAL, the handling of such inferences is incorporated into the type combinators which make new definitions. When a combinator constructs a definition, the combinator procedure determines the valid subsumption inferences from the definition; these subsumptions are then expressed in the lattice. Handling inference at definition time places a constraint on adding new types to the lattice: *new subsumption relations must never be defined between existing types in the lattice, but only between a newly defined type and existing types.*

We cannot, after having defined [Integers] below [Numbers], turn around and place [Integers] below [Strings]. In addition to simplifying the implementation, it allows a theoretical simplification: the lattice of types can now be viewed as a complete subgraph of some hypothetical complete lattice which contains all possible definitions. This would not be possible if the arbitrary addition of subsumption relations were allowed at any point.

The making of type inferences is sometimes a tricky matter; often the inferences are counter-intuitive and involve significant 'lattice combing' to find unexpected generalizations and specializations. The general problem is computationally intractable; TYPICAL takes the middle course of making a useful subset of the possible inferences. The next section introduces some of the problems of combinator inference by describing TYPICAL's 'test suite.' This test suite is a set of definitions and test cases which determine whether or not TYPICAL's type inference procedures are working.

2.2 Example: The Test Suite

TYPICAL defines a test suite of definitions and predicates on those definitions which determine whether or not TYPICAL inference procedures are working appropriately. In particular, they test the inference procedures for intersections, unions, 'image constraints,' and tuple definitions.

The test suite initially describes a set of seven types by using the `PRIMITIVE-SET-OF` combinator; this combinator takes a list of objects and a generalization and constructs a

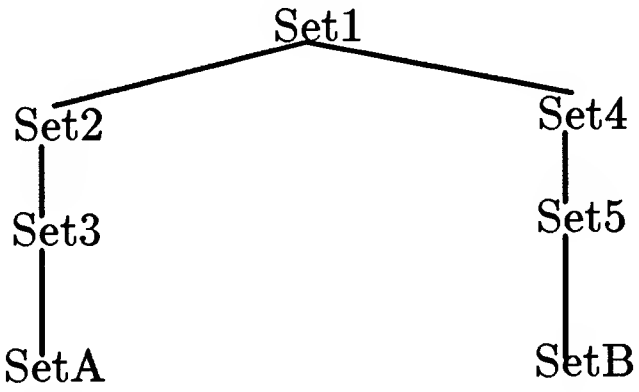

Figure 2-3. The initial lattice for the test suite consists of two inheritance 'towers' underneath a linking set.

type predicate under the given generalization which is satisfied *only* by the given objects. So the initial set of the test suite are defined thus:

```
(define set1 (primitive-set-of '(1 2 3 4 5 6 7 8 9)
             lisp-objects))
(define set2 (primitive-set-of '(1 3 5 7 9) set1))
(define set3 (primitive-set-of '(1 3 5 7) set2))
(define setA (primitive-set-of '(1 3) set3))
(define set4 (primitive-set-of '(1 2 3 4 5 6 7 8) set1))
(define set5 (primitive-set-of '(1 2 3 5 7) set4))
(define setB (primitive-set-of '(1 2 3) set5))
```

This network appears in Figure 2-3; to test the lattice inference algorithms we will construct a 'sandwich' of intersections and unions about Set3 and Set5. Then when we create a 'filling' intersection and union of Set3 and Set5, this will have to be between the unions and intersections defined above and below. The following SCHEME procedure tests intersections:

```
(define (test-intersections)
  (let ((high-type (<AND> set2 set4)))
    (let ((low-type (<AND> setA setB)))
      (let ((sandwich (<AND> set3 set5)))
        (if (not (and (subsumed-by? low-type sandwich)
                      (subsumed-by? sandwich high-type)
                      (subsumed-by? low-type high-type)))
            (ERROR "Intersection test failed."))))))
```

In running the above example, TYPICAL must make the following inferences:

- Since any element in the intersection of Set3 and Set5 must be in both Set3 and Set5,

- And any element in [Set3] must be in [Set2],
- And any element in [Set5] must be in [Set4],
- And any element in both [Set2] and [Set4] must be in the intersection of [Set2] and [Set4],
- Then any element in the intersection of [Set3] and [Set5] must be in the intersection of [Set2] and [Set4]. This implication is the same as saying that the predicates determined by the intersection of [Set3] and [Set5] is subsumbed by the intersection of [Set2] and [Set4].

Therefore, the sandwich must be subsumed by the high type; alternatively, the same reasoning follows for the low type being subsumed by the sandwich. We do the same sort of test to determine if the union combinator is making the appropriate inferences. We union [Set2] with [Set4] and [Seta] with [SetB] to make the high and low types of the sandwich and union [Set3] with [Set5] to make the filling:

```
(define (test-unions)
  (let ((high-type (type-union set2 set4)))
    (let ((low-type (type-union setA setB)))
      (let ((sandwich (type-union set3 set5)))
        (if (not (and (subsumed-by? low-type sandwich)
                      (subsumed-by? sandwich high-type)
                      (subsumed-by? low-type high-type)))
            (ERROR "Union test failed."))))))
```

The procedures for making inferences about intersections and unions are described in Section 4.1.2; they are too involved for this introduction. In the next section, however, we describe the simpler inferences of TYPICAL's `image-constraint` combinator.

2.3 Image Constraints

Image constraint types use a function and a specialization of the function's range to define a specialization of the function's domain. For instance, the type of lists starting with integers is the specialization of lists for which the mapping `CAR` satisfies the type [Integers]. The inference we must make about image constraints is that if the actual constraint of one image constraint is beneath another in the lattice, the corresponding image constraints are beneath one and other. For instance, we wish to infer that the type for lists starting with integers is a specialization of the type for lists starting with numbers.

Image constraint types are generated by a combinator procedure which takes a mapping and a type as parameters. The resulting type is satisfied by objects for which the result of applying the given mapping satisfies the given type. Note that while many mappings used for defining image constraints are extractors or simple functions (like `CAR` or

VECTOR-TWELFTH), mappings may in fact be arbitrarily complicated procedures, consing new structure when presented with objects to map into some other space.[3]

Inferences about the placement of image constraints in the lattice rummage through image constraints defined on the same mapping, looking for those types whose image constraints are directly related to the image constraint of the type being generated. For example, the type of lists starting with integers should be below the type of lists starting with complex numbers. When there are no such types, the domain of the mapping is used as a single generalization. To support this, each mapping has an explicitly declared domain and range; these are defined with the DECLARE-MAPPING! procedure:

```
;;; CAR is a mapping from CONSES into arbitrary objects.
(declare-mapping! car conses lisp-objects)
```

Once defined, they can be used for calls to the IMAGE-CONSTRAINT combinator procedure:

```
;;; Define the type for lists whose CAR is an integer.
(define lists-starting-with-integers (image-constraint car integers))
```

TYPICAL's test suite determines that the image constraint inference procedures are working correctly by constructing — as for intersections and unions — a sandwich which tests the procedure in both directions:

```
(define (test-image-constraints)
  (let ((lists1 (image-constraint car set1)))
    (let ((lists3 (image-constraint car set3)))
      (let ((lists2 (image-constraint car set2)))
        (if (not (and (subsumed-by? lists2 lists1)
                      (subsumed-by? lists3 lists2)))
            (ERROR "Image constraint test failed."))))))
```

2.4 Composing Combinators

Combinators like conjunction, disjunction, power-set or image constraint are *primitive*; they directly construct types and are the units about which subsumption inferences are made. We can compose calls to these primitive combinators to build composite types. Composing image constraints with intersections and unions specifies particular subtypes of various structured objects. We might define the type of integer pairs thus:

```
(<AND> (image-constraint car integers)
       (image-constraint
          cdr (<AND> (image-constraint car integers)
                     (image-constraint cdr empty-lists))))
```

[3]For instance, in the CYRANO program, some mappings carry objects of a type T into the equivalence partitions (which are represented by types) determined by some relation over T. Computing this mapping may require the construction of a new partition (type) as the result of the mapping.

This defines the type of lists whose CAR is of type Integers and whose CDR is of a type which demands that *its* CAR be an integer and *its* CDR an empty list (an object satisfying Empty Lists).

Since combinators are procedures, new *composite* combinators can be defined by defining procedures which call other combinators internally. The above definition of pairs of integers nests procedure calls; we could write a recursive procedure MAKE-CROSS-PRODUCT which could serve as new combinator for defining arbitrary lists of type constrained elements:

```
(define (make-cross-product element-constraints)
  (if (null? element-constraints) empty-lists
      (<AND> (image-constraint car (car element-constraints))
             (image-constraint
               cdr (make-cross-product (cdr element-constraints))))))
```

allowing expressions like:

```
(define integer-pairs
  (make-cross-product (list integers integers)))
(define points (make-cross-product (list reals reals)))
(define notes
  (make-cross-product
    (list (primitive-set-of '(A B C D E F G) lattice-top)
          (primitive-set-of '(1 2 3 4 5 6 7 8) lattice-top))))
(define triads (make-cross-product (list notes notes notes)))
```

Composite combinators can be used to defined new combinators. The <AND> combinator used above is a composite combinator which uses the primitive combinator procedure TYPE-INTERSECTION. TYPE-INTERSECTION takes two types and returns their intersection; <AND> takes a list of types and calls TYPE-INTERSECTION recursively down the list. We could define <AND> as:

```
(define (<AND> type . with-types)
  (if (null? with-types) type
      (type-intersection type (apply <AND> with-types))))
```

TYPICAL is used by **Cyrano** to define new concepts; a key notion in **Cyrano** is the *abstraction function* which maps some space of objects (sometimes types) into a space of types. These functions are essentially combinators; much of **Cyrano**'s progress is in the definition and selection of new abstraction functions which define new domains of operation. Thus, the composable nature of TYPICAL's combinators is an important component of **Cyrano**'s design.

Chapter 3
TYPICAL Implementation

TYPICAL is implemented in SCHEME [RC86], a streamlined dialect of LISP which emphasizes economy of mechanism and simplicity of metaphor. SCHEME is lexically scoped and encourages widespread use of procedures as first class data objects; it is used as the computational teaching tool in MIT's introductory computer science course, 'The Structure and Interpretation of Computer Programs' [AS85]. TYPICAL is written largely in the [RC86] standard SCHEME and was developed in 'C-Scheme' [4] on Hewlett Packard 'Bobcat' computers; C-Scheme (and TYPICAL) also run on a range of other Unix and VMS based machines. TYPICAL can also run (with some work) under other SCHEME implementations and under Common LISP by virtue of a SCHEME compatability package. Copies of TYPICAL are available from the network and mail addresses given in Appendix A-3.

Scheme was chosen for reasons of elegance and portability. SCHEME more naturally expresses higher order functions and the use of functions as objects than other LISP dialects. On a more pragmatic note, the lack of advanced user interface facilities and other 'hair' in the development environment kept the headier temptations of programming from the author; in most work in Artificial Intelligence, I fear, user interface specification happens far too soon. Scheme, with its avoidance of half-baked interface metaphors, forces programmers to get the content and 'working metaphors' right and *then* develop the user interface. In fact, the development of TYPICAL has suggested just such a user interace, where the interface is specialized by extensions to TYPICAL's lattice.

In the following sections, we introduce the implementation of TYPICAL. This chapter describes the structures used to implement types and the general combinator mechanism

[4] C-Scheme is a version of MIT-Scheme implemented in C for Unix and VMS machines. Information about C-Scheme availability is given in Appendix A-3.

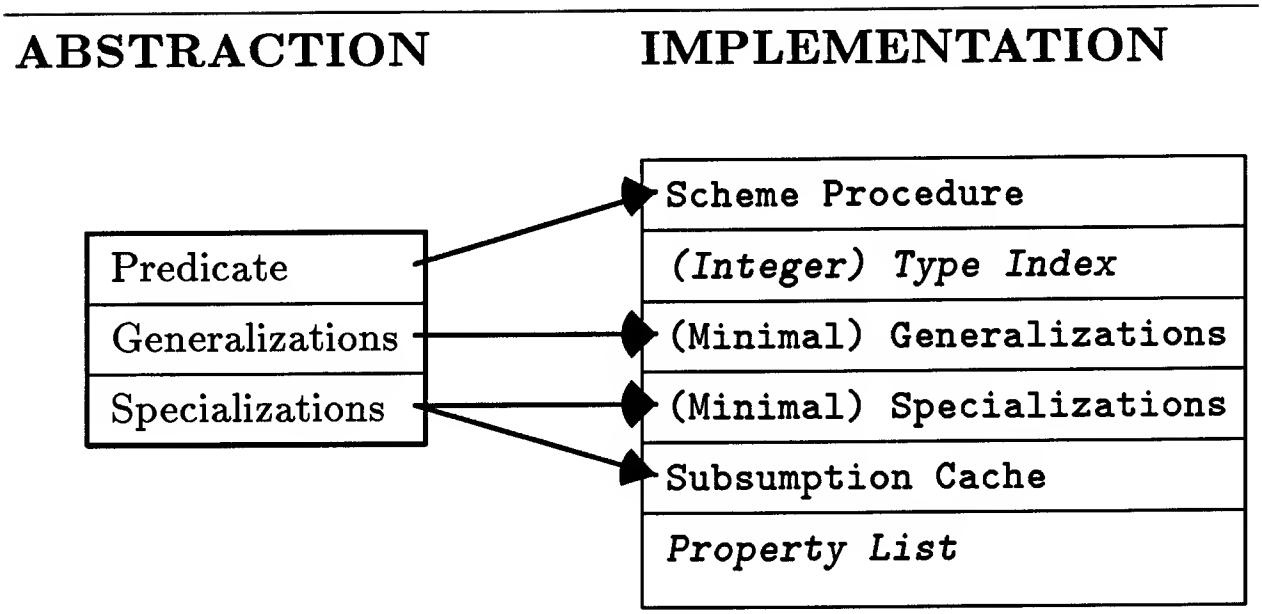

Figure 3-1. The concrete implementation of type descriptions stores some information redundantly, while factoring some easily computed information out of individual descriptions. The italicized slots on the right hand side are additions to the implementation outside of the predicate/lattice abstraction on the left.

used to generate them. (The details of particular combinators are described in Chapter 4 and Appendix A-1.) Finally, the implementation of undetermined values is briefly introduced and discussed.

3.1 Type Descriptions

TYPICAL uses a data structure called a *type description* to describe predicate procedures implemented in SCHEME; the data structure is annotated with information about the predicate. The abstract and concrete implementation of type descriptions is depicted in Figure 3-1. Abstractly, a type description has 3 properties:

1. A characteristic predicate for the type. This is a test which determines whether or not a given instance satisfies the type. Strictly speaking, this does not have to be a predicate, since it can return a special undetermined 'i don't now' token as well as true or false.
2. A set of generalizations. These are the types for which satisfaction is entailed by the satisfaction of this type.
3. A set of specializations. These are the types for which satisfaction entails satisfaction of this type.

As implemented, the representation is more complicated; largely for efficency reasons, some information is stored redundantly, while some redundant (and easily computed)

information is factored out of individual descriptions. The actual structure implementing type descriptions is a SCHEME vector with 6 elements:

1. The characteristic predicate for the type. This is generally (except in rare cases, such as primitive types) a closed procedure whose closed variables bind the types or terminals from which the type was generated.
2. A unique integer index for the type. Each type has a unique integer ID, ascending from zero with each new type definition. This index is useful for canonically ordering lists of types or for mapping types into a linear sequence such as the elements of a bit vector. It is also useful, interactively, for getting a handle on a type you have seen printed.
3. The immediate generalizations of the type. This does not contain all the generalizations of the type, but merely the most specific subset of the type's generalizations. The generalizations of types, as actually stored, constitute a minimal generator of the type's generalizations; the recursive closure of this tree would produce the complete set of generalizations.
4. The immediate specializations of the type. As for generalizations, this is not a complete set, but rather a minimal set from which the complete set may be generated. Rather than storing all of the specializations of the type, only the most general subset of the specializations is stored.
5. A bit-vector which caches subsumption relations in the lattice. The index of a type is used to offset into this bit-vector; if the corresponding bit is on, the type is subsumed by the type which corresponds to the index. In keeping this cache on each type, TYPICAL's implementation trades off time for space; it provides for constant time subsumption queries by using an $O(n^2)$ bit table for caching subsumption information.
6. A table of incidental properties of the type. This is used by some TYPICAL combinators in making inferences about subsumption relations. It is also avaiable to systems (like **Cyrano**) which use TYPICAL as a representation language: types can be annotated with program specific information such as known examples, sources of definition, etc.

If our abstraction for types supports only the predicate `SUBSUMED-BY?`, type descriptions are immutable objects. The contract of the lattice demands that the definition of new types have no effect on subsumption relations between existing types. If the interface is extended to include accessors for the stored generalizations and specializations, such immutablity vanishes as immediate generalizations are added and subtracted to reflect the presence of new types and the minimization of the represented lattice.

3.2 Combinators

New types are defined by an extensible combinator language which combines either existing types or primitive terminals to construct new types. The existing types combined in this

way are all the result of previous combinator invocations. The primitive terminals so combined are LISP objects of various sorts; most commonly they are explicit lists or opaque procedures. In the previous chapter we saw three sorts of primitive terminals being used: test functions, mapping functions, and fixed lists of elements.

The use of procedures to define new types gives the power of procedure application to the definition of new constraints, classes, and concepts. Calls to combinators can be nested, passed as arguments, or combined by higher-order functions. By maintaining the power of the procedure call model, TYPICAL inherits or assumes the semantics of SCHEME, allowing other packages to integrate with TYPICAL's facilities.

A given combinator invocation produces both an implemented predicate and a position for that predicate in the subsumption lattice. This position is determined by a list of generalizations and a list of specializations. These three results are produced by applying procedures given in the combinators definition. For instance, we define the combinator `PRIMITIVE-SET-OF`:

```
;;; The PRIMITIVE-SET-OF combinator takes a list of elements and
;;; a generalization and returns a type underneath the given generalization
;;; which is only satisfied by members of the given list.
(define primitive-set-of
  ;; Parameters: (elements generalization)
  (type-generator
   ;; The predicate calls MEMBER on the specified elements.
   (lambda (elements generalization) (lambda (x) (member x elements)))
   ;; The generalizations of a primitive set are the single one given.
   (lambda (elements generalization) (list generalization))
   ;; We don't define any specializations.  A more sophisticated
   ;;  version of this function might look for other primitive sets
   ;;  whose elements were subsets of this one.
   (lambda (elements generalization) ())))
```

The TYPICAL procedure `TYPE-GENERATOR` is a higher-order function used to define combinators from the individual methods which generate or infer a new type's properties. In the example above, the predicate for a `PRIMITIVE-SET-OF` type is a closed lambda-expression which calls the SCHEME function `MEMBER` on a potential instance and the list passed in the original combinator invocation. The generalization of a newly created `PRIMITIVE-SET-OF` type is simply the existing type given in the initial specification. Finally, the newly created type has no known specializations.

When a combinator is actually invoked, it's inference procedures compute the generalizations and specializations of the type to be constructed; if the resulting generalizations and specializations are identical, the type is tautologically equivalent to this common specialization and generalization. In this case, the combinator call returns this equivalent type. If the specializations and generalizations are distinguished (as is usually the case),

the predicate generator procedure is called to construct a SCHEME predicate. This procedure is then sotred in a newly constructed type description.

At this point, the type description is installed in the lattice between the generalizations and specializations computed before. This is the only time when new links are added to the lattice.[5] Once the type description has been installed in the lattice, it is returned as a result of the combinator.

Combinators *memoize* their results; if a combinator is called repeatedly with identical arguments, it will only generate a new type once, recording and returning the first-time result on all subsequent calls with the same arguments. This assures identity of definitions as well as the avoidance of wasted effort in redefining existing types.

3.3 Representing Uncertainty

In the previous sections we referred to the procedures generated by combinators and attached to types as 'predicates.' Types in TYPICAL actually describe three-valued functions which may return an undetermined 'i don't know' as well as boolean true or false. When a predicate returns 'i don't know' for an object we say that the type is *undetermined* for the object. Thus, for a given object and type in a lattice, the type may be satisfied, unsatisfied, or undetermined.

The representation of uncertainty does not deeply effect the structure of the lattice; subsumption in the lattice still indicates predicate entailment. $Undetermined(x, T)$ makes no claim about x and the generalizations of T, just as $Unsatisfied(x, T)$ makes no claim about the satisfaction of any of T's generalizations. On the other hand, the representation of uncertainty does effect the generation of predicates; each predicate cannot simply use SCHEME's boolean combining forms, since by SCHEME's semantics the non-nil "i don't know" token is logically true. The way TYPICAL combines three-valued predicates is addressed in the next chapter, where TYPICAL's basic combinators are presented and explained.

[5]There are certain carefully controlled violations of this principle. One particular case, described in Section 4.5 (Page 33), is in the definition of recursive types which have as inferred generalizations and specializations other types which refer to the recursive type itself.

Chapter 4
TYPICAL Combinators

This chapter presents the basic combinators provided by TYPICAL. These are defined by `TYPE-GENERATOR` definitions much like those we saw in the previous chapter. I will detail the methods used to generate predicates and make inferences about relations in the lattice. The semantics and justification of these methods will be informal. Appendix A-1 introduces a more formal semantics for TYPICAL's basic combinators and demonstrates the soundness of the inference methods used by them.

Types defined by TYPICAL fall into two broad categories: *analytic* and *synthetic*. Analytic types are types defined in terms of other types; synthetic types are types defined 'empirically' in terms of LISP predicates or enumerated sets. New type descriptions are generated in a combinator language which create either composite definitions (analytic types) or primitive definitions (synthetic types).

Types are described by type descriptions which are themselves LISP objects subject to type classification; Figure 4-1 is a fragment of the lattice of meta-types beneath the type Types. As indicated, types naturally fall into two roughly epistemic categories: *analytic* and *synthetic*. **Analytic** types are defined in terms of other types; **synthetic** types are defined in terms of enumerated sets or opaque LISP predicates.

Analytic types are further classified as either *direct* or *indirect* depending on how the apply the types by which they are defined. **Direct types** apply the types they combine directly to the objects they test: an intersection is satisfied if an object is in both of the types it combines, a union is satisfied if it satisfies either of them, and a complement is satisfied if it *doesn't* satisfy the type it complements. **Indirect** types, on the other hand, transform their subjects into another space before using the types they are defined in terms of. For instance, **image constraint** types carry their objects through a particular mapping before applying their test constraint.

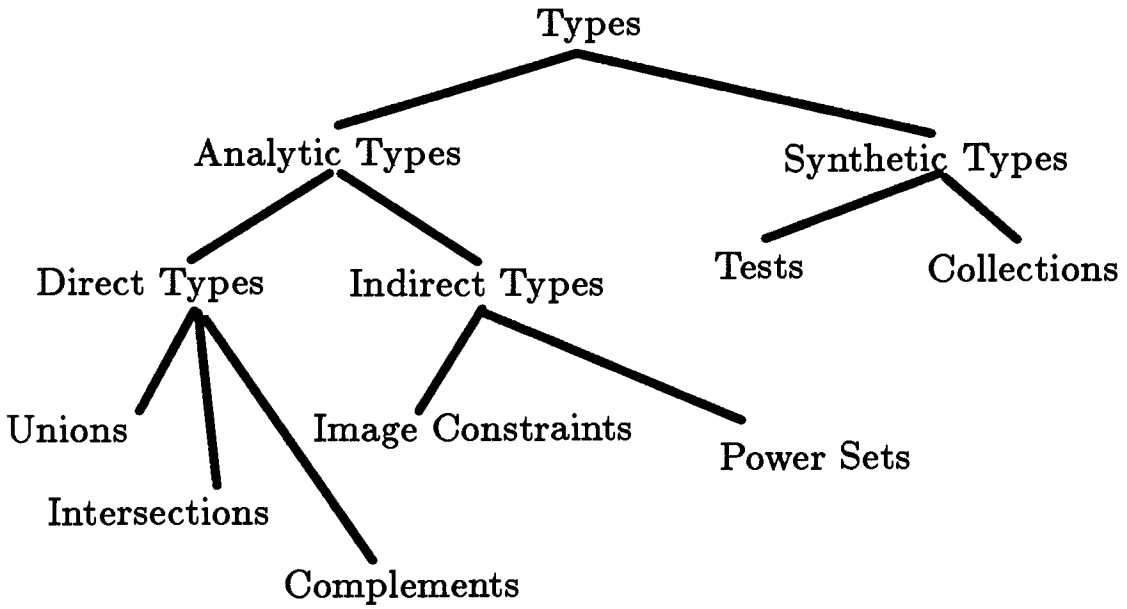

Figure 4-1. The taxonomy of types is expressed in the lattice of types.

Synthetic types also fall into two disjoint categories: *tests* and *collections*. **Tests** are types determined by opaque LISP predicates and generally describe the "natural" environment of the implementation. **Collections** are sets of objects, fixed or mutable, which are in practice enumerable.[6] TYPICAL provides a generic function for returning the elements of a collection; the applicability of this function defines the distinction between tests and collections.

Orthogonal to the distinction between analytic and synthetic types is a distinction — arising from the representation of mutable or incompletely specified types — between *complete* and *partial* types in the lattice. For practical reasons, type satisfaction is not simply a binary distinction; it may be possible for type satisfaction to be undetermined for a given object. This is especially true of empirical properties recognized by **Cyrano**; satisfaction or non-satisfaction of an empirically determined type may have to wait upon evidence or counterevidence from the progress of the program. A **complete** type represents a two-valued predicate returning 'true' or 'false'. A **partial** type represents a three-valued predicate which may also return the ignorance marker 'i don't know'.

Partial types are useful (in the program **Cyrano**, for instance) for representing types which are actually or pragmatically undetermined for certain objects. A class of empirical observations, for instance, is actually undetermined; a new phenomenon could be added to the class at any point. Even if we tagged all phenomena with 'time tags,' we would simply

[6]On a finite machine, all the types in the lattice are in theory finitely enumerable, but synthetic collections have the distinction of being practically enumerable; there is a provided procedure for accessing the elements.

divide the class into a complete class of past events and a partial (and currently empty!) class of future events.

Furthermore, some types, while formally complete, may be pragmatically undetermined. Evidence for or against some completely specified empirical property (e.g. that some relation is an equivalence relation) must be accumulated; while this accumulation occurs, the property is undetermined for the object (e.g. the particular relation), despite a precise criterion for confirmation or at least disconfirmation of the property. It was this pragmatic indeterminancy which originally motivated the representation of uncertainty in TYPICAL.

Synthetic types are inherently partial or complete; analytic types inherit their partial or complete status from the types they are defined in terms of. For test types, completeness is determined by the predicate for the test; if the predicate can return an ignorance token, then the corresponding type is partial. Collection types are more complicated because of the presence of mutable collections. A fixed collection is complete, but a mutable collection is complete only if objects are only added *on construction.* In TYPICAL, a complete but mutable collection is called a *generated collection*; generated collections have associated *generators* which are the only way new members can be added to the collection. Mutable collections which are not generated collections are inherently partial because they are undetermined for all current non-members.

4.1 Direct Types

TYPICAL defines three sorts of direct types: unions, intersections, and complements. Union types are disjunctions of other types; an object satisfies a union type if it satisfies any of the types unioned. Intersection types are conjunctions of other types; an object satisfies an intersection type if it satisfies all of the types intersected. Complement types are satisified if the type the complement is *not* satisfied.

The primitive combinators `TYPE-INTERSECTION` and `TYPE-UNION` are use to do binary intersection and union of other type definitions. The composite combinators `<AND>` and `<OR>` do n-ary intersections and unions by repeatedly invoking `TYPE-INTERSECTION` and `TYPE-UNION`. Below we describe only the workings of binary intersection and union; n-ary combinations are simply implemented by repeated applications of the binary combinations.

4.1.1 Predicate Functions of Direct Types

Generating new predicate functions for direct types is not particularly complicated; the only subtlety is introduced by the presence of ignorance tokens. Normally, generated predicates would simply logically combine or invert the satisfaction results of the types they were generated from. For instance, the complement of a type P would have a predicate which checks if its argument satisfies P, failing if it does and succeeding otherwise. The

$\wedge$	T	F	?		$\vee$	T	F	?		$\neg$		
T	T	F	?		T	T	T	T		T	F	
F	F	F	F		F	T	F	?		F	T	
?	?	F	?		?	T	?	?		?	?	

Figure 4-2. The semantics of unions, intersections, and complements can be described by three-valued truth tables. T represents a type satisfied, F a type unsatisfied, and ? a type undetermined.

difficulty appears if the satisfiability of P is undetermined. In the case of complementation, indeterminancy must noted and passed on; unfortunately, in SCHEME (or LISP), if the logical sense of the ignorance token were simply inverted, ignorance (represented by a returned ignorance token) would become falsity, indicating certain denial. Since ignorance tokens exist, logical combination becomes three-valued rather than two-valued.

Figure 4-2 shows a three-valued truth table for union, complementation, and intersection. A union is satisfied if either of its component types are satisfied; it is unsatisfied if both of its component types are unsatisfied; otherwise, it is undetermined. An intersection is satisfied if both of its component types are satisfied; it is unsatisfied if either of its component types are unsatisfied; otherwise, it is undetermined. Finally, a complement is satisfied if its component type is unsatisfied; it is unsatisfied if its component type is satisfied; and it is undetermined if its component type is undetermined.

4.1.2 Subsumption Inferences of Intersections and Unions

Computing the generalizations and specializations of direct types is not as straightforward as generating predicate procedures. Section A-1.3 (Page 79) shows that, in general, computing subsumption is — even with only AND and OR – computationally intractable. To keep the time required for inferences manageable, TYPICAL uses polynomial-time algorithms which, while sound, are not complete; TYPICAL finds only a subset of the valid subsumption relations for a newly created type. This section informally describes those algorithms.

The obvious generalizations and specializations of an intersection or union are the types they union or intersect. An intersection is below the nodes it intersects; a union is above the nodes it merges. By the transitivity of subsumption, a type so placed will also lie under or above all the generalizations or specializations of these nodes.

TYPICAL goes a step further to infer relations to a class of types called *accidental merges* which lie above or below a new intersection or union. For any pair of nodes merged (in either lattice direction: union or intersection) in the lattice, there may be other merges (intersections or unions) above or below them which should be connected to the

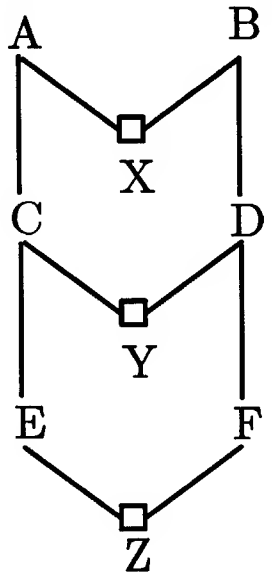

Figure 4-3. Accidental merges in the lattice form a class of subsumption inferences which can be made in polynomial time. If 'down' is the direction of intersection, Y should be directly below X and directly above Z. When Y is defined, these generalizations must be found in the lattice around the new definition.

newly created merge. Figure 4-3 illustrates the problem in computing subsumption. By convention, up will be in the direction of generalization; hence X is $A \wedge B$, Y is $C \wedge D$, and Z is $E \wedge F$. The problem is that by these definitions, X should be beneath Y and Y should be beneath Z; the problem for a subsumption algorithm will be to find — given two nodes like C and D being merged into Y — the merges X and Z. Logically, given:

$$(E \longrightarrow C) \wedge (C \longrightarrow A)$$
$$(F \longrightarrow D) \wedge (D \longrightarrow B)$$

represented in a lattice of implication (subsumption), we wish to derive the implications (subsumptions):

$$(E \wedge F) \longrightarrow (C \wedge D)$$
$$(C \wedge D) \longrightarrow (A \wedge B)$$

Combinations like $(E \wedge F)$ or $(A \wedge B)$ are indirectly related to $(C \wedge D)$ and are called 'accidental merges'. Accidental merges may look familiar; the **test-suite** example of Section 2.2 (Page 10) tested TYPICAL's algorithms for finding V-Merges and M-Merges.

Given a lattice direction, accidental merges are of two sorts: M-merges and V-merges. In Figure 4-3, the merge X is an M-merge and the merge Z is a V-merge. For intersections, M-merges are above the intersection and V-merges are below it. For unions, directions are inverted, and M-merges are below the union while the V-merges are below it. Since the processes for intersections and unions are mirror images, we will describe only intersection,

```
To find M-merges above and(a,b):
   Make a set of marked nodes M;
   Make a set of m-merges J;
   For every superior s of a or b,
      process the node s;
   To process a node n:
     add n to a set of marked nodes M;
     for each inferior merge i of n:
         if every superior of i is in M  (i.e. is marked),
            add i to J and mark the node i;
```

Figure 4-4. The algorithm for finding M-merges of two nodes marks all the generalizations of two nodes and looks beneath them for nodes which merge marked nodes.

noting that unions may be handled by simply replacing 'up' with 'down' in the algorithm. For the interested reader, Appendix A-1 presents descriptions and soundness proofs for both intersections and unions.

To find V-merges we search the lattice below one of the types being combined for types which are beneath the other type being combined. In the example of Figure 4-3, to find V-merges beneath X, we descend the lattice beneath A looking for nodes which are beneath B.

Finding M-merges is more complicated. We can look at finding M-merges as a problem of finding V-merges from 'the other side'. An M-merge is a V-merge of two 'superiors' (these are the generalizations in the case of intersection, the specializations in the case of union) of the nodes being combined. A simple algorithm would try to find the V-merges of all possible pairs of the two nodes' superiors. Since the space of these pairs is probably quite large, we would like to interleave the various searches for V-merges. We use a marker propogation algorithm which marks all of the superiors of the nodes being merged and then looks down from them for potential M-merges. Given the marking of superiors, an M-merge is any node which merges marked nodes or other M-merges. The algorithm (given in Figure 4-4) ascends the lattice, marking each superior and checking for M-merges directly beneath it (these are merge nodes which have all of their superiors marked). Each discovered M-merge is marked, and *its* inferiors are checked for M-merges. (This step captures nested M-merges which merge other M-merges.) This will find all M-merges because there must always be a 'final' marked superior which will make a node an M-merge; when this final superior is marked, the M-merge beneath it will also be discovered.

4.1.3 Subsumption Inferences for Complements

TYPICAL makes no use of complementation or disjointness information in making subsumption inferences between intersections or unions in the lattice. In determining the subsumption relations of a newly defined complement, however, TYPICAL does seek to infer its relation to other known complements.

The basic scheme of these inferences are quite simple; the generalizations of a complement type $\neg T$ are the defined complements of the specializations of T. It's specializations are the defined complements of the generalizations of T. To find these types, we search below T for types with defined complements; any such complements are generalizations of the complement type $\neg T$. In the same way, we find the specializations of $\neg T$ by looking for defined complements of the generalizations of T.

As will be shown in Section A-1.1.1.5 (Page 71), this algorithm is — by itself — complete; if the rest of the lattice is complete, this algorithm will make all the valid inferences about the complement's relation to other types. Unfortunately, the other combinators in TYPICAL are provably incomplete, so this result is not decisive.

4.2 Indirect Types

TYPICAL implements two sorts of indirect types: *power sets* and *image constraints*. **Power sets** are meta-types: the power set of the type **Integers** is the type satisfied by all specializations of **Integers**. **Image constraints** are types which constrain some actual or virtual component of an object: the **CAR** of a list, the first element of a vector, or the canonicalization of a type description.

4.2.1 Power Sets

The predicate function for a power set is simply a closed call to the TYPICAL procedure SUBSUMED-BY?. Since new subsumption relations are never created between existing types in the lattice, power set types are always *complete* in the sense defined by Section 4; they return a true or false which holds for all time.

Computing the generalizations and specializations of a power set is also straightforward: ascend and descend the lattice from the defining type, collecting known power sets along the way. Since relations between existing types in the lattice are fixed, no new relations between powersets will be introduced; and when a new power set is created, it will take into account all the existing power sets. A mapping between a type space and the corresponding power-set space is shown in Figure 4-5. The power set of the integer type is placed between the power set of the number type and the power set of the natural-number type, since natural numbers are below integers in the lattice and general numbers are above them. The power sets of reals and complex numbers, as yet undefined, are not part of the

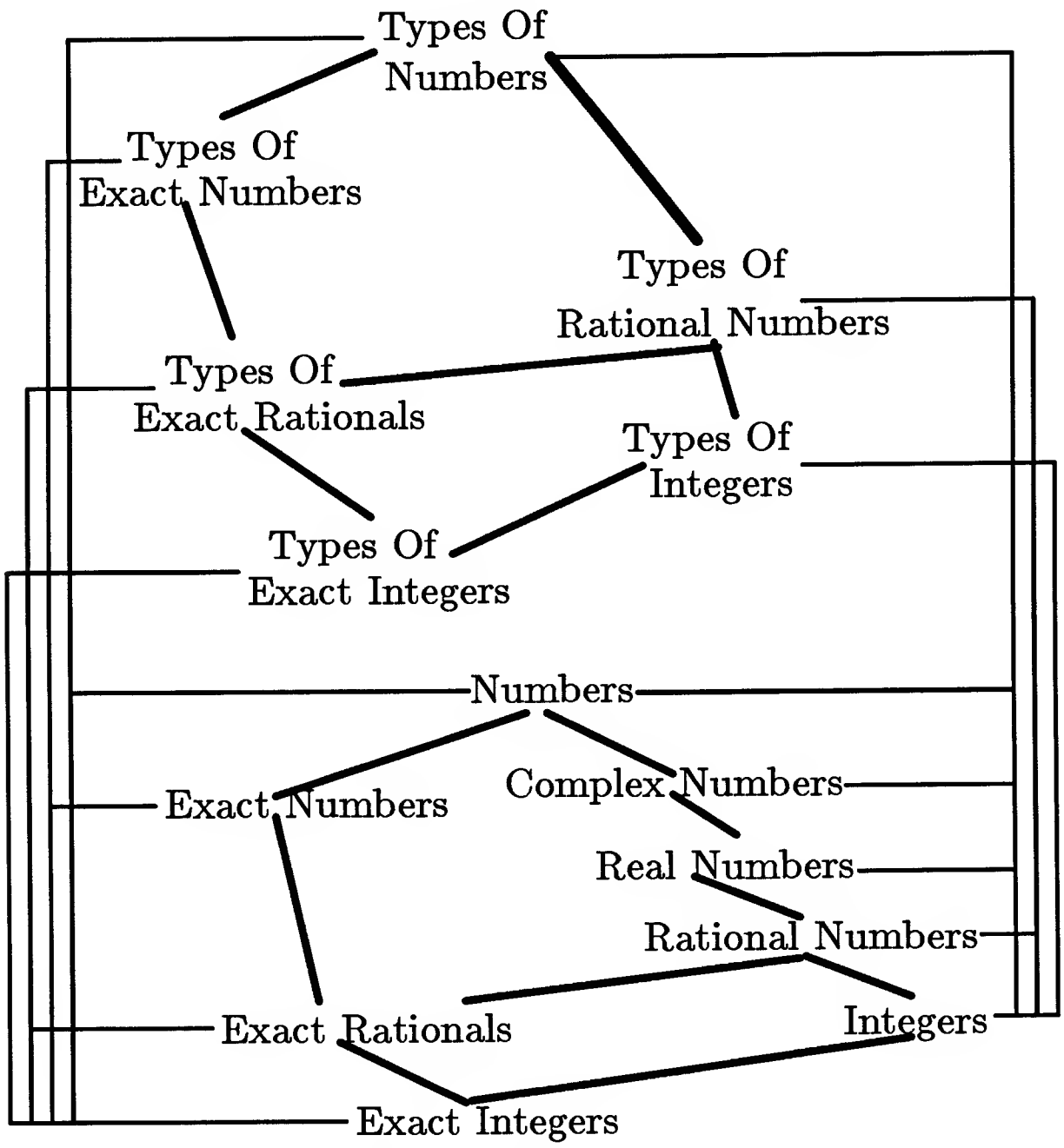

Figure 4-5. The structure around a type is indirectly reflected in the structure around its power set. The bold lines in the figure indicate subsumption in the lattice, while the finer lines indicate satisfaction of types by metatypes.

picture. When they are defined, they will be appropriately integrated into the power-set space beneath $PowerSet(\boxed{\text{Numbers}})$.

4.2.2 Image Constraints

The other sort of indirect type, the image constraints, allows definitions of types which predicate on components of structured objects. These parts may be actual parts: slots in a record structure or bindings on a property list. They may also be virtual parts: the results of inheritance or some other non-trivial computation. We will indicate the image constraint defined by C on a mapping f by $\boxed{\xrightarrow{f} C}$; this image constraint type is satisfied by some x if and only if $f(x)$ satisfies C.

The predicate function for an image constraint calls the mapping f as a function and then checks if the result satisfies P. An image constraint is a partial type (as in Section 4 above) if its predicate is partial.

Computing the generalizations and specializations of an image constraint is much the same as for a power set. The algorithm simply ascends and descends the lattice from the constraint on the image, looking for specializations or generalizations which constrain the same image. If no specializations are found, there are none; if no generalizations are found, the domain of the mapping is used as the constraint's generalization.

The mappings used by an image constraint must be declared as mappings, and their ranges and domains made explicit. Sometimes these mappings are actual components of an implemented structure; at other times they are virtual components accessible after a significant computation. For instance, when relations are represented as types of pairs, the inverse of a relation could be described by the image constraint of the relation (as a type of pair) on a 'twist' function which switches the left and right halves of the pair.

4.3 Composite Combinators

TYPICAL also provides a number of composite combinators which combine image constraints with intersections. Two are worthy of particular notice, `RECORD` and `CROSS-PRODUCT`. The `RECORD` combinator is for defining composite constraints on structured data. Its arguments are an alternating list of mappings and types; the type it returns is the conjunction of the image constraints generated for each mapping and type. For instance, suppose we define the type $\boxed{\text{People}}$ and the mappings `AGE`, `NATIVE-TOUNGE`, and `SEX`. We could then use `RECORD` to define the type of 'young' natively english-speaking men:

```
(define young-english-speaking-men
  (record age less-than-30-years
          sex (primitive-set-of '(MALE) sexes)
          native-tounge (primitive-set-of '(ENGLISH) languages)))
```

`RECORD` is particular useful for describing subtypes of record-like data structures. One natural 'programming language' extension of TYPICAL would be a record defining form which autmatically declared some of its accessors as appropriately typed mappings.

Another important composite combinator is CROSS-PRODUCT. A simple implementation of this was illustrated in Section 2.4. The CROSS-PRODUCT combinator takes an arbitrary number of arguments, each of which constrains the corresponding element of a list. CROSS-PRODUCT is useful for typing unstructured combinations of instances of various types; for instance, pairs of coordinates, triplets of people, or quartets of musicians:

```
(define points (cross-product integers integers))
(define possible-menage-a-trois
  (cross-product people people people))
(define singing-quartets
  (cross-product bass tenor alto soprano))
```

4.3.1 Unwinding Composite Combinators

When intersections and image constraints are merged, the actual constraints on an individual mapping are often lost in the far reaches of the lattice. The TYPICAL function DETERMINE-IMAGE-CONSTRAINTS recovers this constraint information from the lattice. Given a mapping and a type, it climbs the lattice from the given type, checking all of its generalizations to see if any are used to constrain the mapping of interest. Finally, it returns all of the actual constraints encountered on the way up. For instance, if we defined the type Computer Dates[7] as follows:

```
(define male-female-pairs (cross-product men women))
(define unmarried-pairs (cross-product unmarried unmarried))
(define computer-dates
  (type-intersection male-female-pairs unmarried-pairs))
```

and called DETERMINE-IMAGE-CONSTRAINTS to determine the constraints on the CAR of the composite type Computer Dates, we would get:

```
(determine-image-constraints car computer-dates)
```
$\Longrightarrow$ (male unmarried)

DETERMINE-IMAGE-CONSTRAINTS climbs the lattice from Computer Dates to find the individual constraints on CAR generated by the cross product combinations Unmarried Pairs and Male/Female Pairs. We could then produce a single type combining all of these constraints by just passing the result to the <AND> combinator. The TYPICAL procedure MAPPING-CONSTRAINT does just that:

```
(define (mapping-constraint mapping type)
  (apply <AND> (determine-image-constraints mapping type)))
```

so we can call MAPPING-CONSTRAINT as above:

```
(mapping-constraint car computer-dates)
```
$\Longrightarrow$ male $\wedge$ unmarried

[7]Confining the model to heterosexual arrangements.

4.4 Synthetic Types

Considering types as forming a grammar, the analytic combinators are the productions of the grammar; they take expressions and combine them to make new composite expressions. Sometimes the expressions so combined are themselves composite, but at some point they must bottom out in the terminals of the grammar. TYPICAL's synthetic types are those terminals and the synthetic combinators are the way of defining new terminals.

Synthetic types fall into two classes: tests and collections. In practice both are ultimately implemented as predicate tests, but collections have the property of *practical enumerability*. This means that there is a function which, given a collection type, will return a list of objects currently in the collection; an object will satisfy the type (providing the type is not modified) if and only if it is in the list of objects returned by this procedure.

The lattice implementation provides a generic function for enumerating instances of a collection type; the applicability of this function determines the difference between collections and tests. The implementation also provides a generic function for adding elements to a set; the applicability of this function determines another division among collections, between fixed collections and mutable collections.

The generalizations and specializations of synthetic types are always provided by the user or program calling the combinator; the lattice implementation does no inference on the definitions of synthetic types. Particular programs generating new synthetic types may invoke considerable calculation to compute the generalizations passed into the lattice implementation, but this is then always used without processing by TYPICAL.

Whether or not a synthetic type is complete or partial cannot be generally determined by TYPICAL. Immutable collections, for instance, are always complete. Simple predicate tests or arbitrarily modifiable collections, on the other hand, carry no such guarantees. The distinction between partial and complete synthetic types is determined (except in the cases mentioned above) by a synthetic collection of 'revealed complete types'. By convention, types are added to this collection only when newly created; hence it is a generated type itself, and complete rather than partial.

4.4.1 Tests

Test types are types whose determining predicate is explicitly and opaquely specified in their definition. Such black box types are used as both experimental 'place-holders' in program development (say, before the mechanism of a new combinator has been completely determined) and as primitives of the implementation from which more complicated composite type combinations are constructed.

One sort of test type combinator — the `SIMPLE-TYPE` combinator — was presented in Chapter 2. Taking a procedure and a type, it defines a subtype of the type with the procedure as its characteristic predicate.

The other sort of test type is the **divided type**, for which satisfaction is determined by two predicate functions: an *in-test* and an *out-test*. If an object passes the in-test, the divided type is satisfied; if it passes the out-test but not the in-test, the divided type is unsatisfied; if it passes neither, the divided type is undetermined for the object.

4.4.2 Collections

Basic TYPICAL defines four sorts of collections: fixed collections, generated collections, empirical collections, and divided collections.

Fixed collections are simply a fixed set of objects; since they are immutable, fixed collections are neccessarily complete. Nothing clever is done about placing fixed collections in the lattice; every fixed collection is beneath a type specified when the type is created. While it might be possible to place these finite sets in the lattice in some intelligent manner (e.g. automatically placing the set $\{1,2,3\}$ beneath $\{1,2,3,4\}$ or the set $\{1\}$ beneath the type Integers), this is not currently done.

A generated collection is a mutable type which is nonetheless complete; only newly generated objects are added to the type. Generated collections are implemented by simple types. The predicate for a generated collection keeps a list of objects satisfying the type; any other objects are summarily rejected. Generated collections are always complete; any objects satisfying it are either recorded or currently unconstructed. It is normally an error to add an object to a generated collection; the procedure `COLLECTION-GENERATOR` takes a generated collection and a generator procedure and returns a new generator procedure which will add results from the given procedure to the collection. Thus, if we created a collection Interesting Ideas beneath the type Ideas:

```
(define interesting-ideas (generated-collection ideas))
```

and had a procedure `REAL-GENIUS` for generating interesting ideas, the expression:

```
(DEFINE GENIUS (COLLECTION-GENERATOR INTERESTING-IDEAS REAL-GENIUS))
```

would define a procedure `GENIUS` just like `REAL-GENIUS` except that its outputs are added to the collection Interesting Ideas. Adding to a generated collection by a mechanism other than a procedure generated by `COLLECTION-GENERATOR` signals an error.

Empirical collections are defined beneath particular types and determine satisfaction based on a finite set of members which may be enlarged arbitrarily. If an object is not in the list of members of the type, its status is undetermined; since it might be added to the set/type later, satisfaction of empirical collections is always positive, and its predicate never returns the false value. Empirical collections are used in **Cyrano** for the definition of classes of 'phenomenon' (e.g. procedure call instances, action/result combinations, etc) for which no 'non-examples' are strictly known.

Divided collections are implemented on top of TYPICAL's divided test types. Divided collections are defined by a modifiable in-set and out-set, which are used by the in

and out functions of a divided test type. Objects in the in-set satisfy the type; objects in the out-set fail to satisfy the type; and objects in neither are undetermined for the type.

Divided collections are used by **Cyrano** to describe the empirical properties of the various domains it explores. The empirical analyses and experiments carried out by **Cyrano** modify divided collections which describe particular empirical properties.

4.4.2.1 Collection Functions

The functions provided by TYPICAL for dealing with collections are `COLLECTION-ELEMENTS` and `COLLECTION-MODIFY!`. The current members of a collection type can be accessed by calling the procedure `COLLECTION-ELEMENTS` on the type. For some collection types, the value returned by this procedure may be modified by the procedure `COLLECTION-MODIFY!`. `COLLECTION-MODIFY!` takes three arguments: an object, a modifiable collection type, and a boolean flag. If the flag is true, the object is added to the collection; if it is NIL the object is removed (or declared out of) the type.

These two procedures operate by referencing two properties — the `ELEMENTS-FUNCTION` and `MODIFY-FUNCTION` properties — of the type.[8] These properties reference procedures which may be called to enumerate or modify the collection type. The presence of these properties determines whether a type is a collection or a collection is mutable. If a type does not have an `ELEMENTS-FUNCTION` property, it is not a collection (by definition); if a collection does not have a `MODIFY-FUNCTION` property, it is immutable (by definition). These properties are defined by the procedures which generate collections; such procedures call the primitive combinators and add properties to that result. Thus, all the collection combinators are really composite combinators; they call primitive combinators within themselves to do type construction and then modify these constructed types by adding appropriate `ELEMENTS-FUNCTION` or `MODIFY-FUNCTION` properties.

4.5 Recursive Types: Inductive Definitions

In all the cases above, the properties of a type — its characteristic predicate, generalizations, and specializations — could be determined without having the generated type present. However, some definitions are naturally recursive. For instance, a list of integers can be defined recursively as either the empty list or a CONS whose `CAR` is an integer and whose `CDR` is a list of integers. Such a type might look like this:

$$\boxed{\text{Integer Lists}} \equiv \boxed{\text{Empty Lists}} \vee \left(\boxed{\xrightarrow{CAR} \boxed{\text{Integers}}} \wedge \boxed{\xrightarrow{CDR} \boxed{\text{Integer Lists}}} \right)$$

Unfortunately, one cannot do this in the framework of TYPICAL's combinators since to define the type one must already have a pointer to it to create the component image constraints. TYPICAL implements a special case of recursive types by breaking the contract

[8] These properties are stored in the table of incidental properties on each type definition.

of TYPICAL in a controlled manner. One element of TYPICAL's contract is the guarantee that once two types have been defined, no later definitions will change the subsumption relation between them. TYPICAL's recursive types — called inductive definitions — constructs a 'simple type' from a recursive SCHEME predicate and then connects this type to other types in the lattice after the `SIMPLE-TYPE` combinator has returned.

By installing relations in the lattice after the type has been generated, inductive definitions violate the contract of the lattice to not place subsumption inferences between existing types. However, if we treate the `INDUCTIVE-DEFINITION` combinator as a primitive combinator in terms of the lattice's contract, the modularity boundaries demanded by the contract remain secure.

An inductive definition combines an *anchor* type, a *test* type, and a list of *links*. Each *link* is a declared mapping acceptable to the `IMAGE-CONSTRAINT` combinator. An object satisfies an inductive definition if it either satisfies the *anchor* or it satisfies the *test* type and each *link* of the object satisfies the inductive definition.

Defining the predicate for an inductive definition is relatively straightforward; it might look something like this:

```
(define (inductive-definition x)
  (if (in? x anchor) #T
      (if (in? x test)
          (every (lambda (link) (inductive-definition (link x)))
                 links)
          #F)))
```

where `anchor` and `test` are types and `links` is the list of link mappings.

In determining the location of an inductive definition in the lattice, TYPICAL makes three sorts of inferences: inferences about the space the definition divides (this is the generalization(s) of the type), inferences about the relation of the type to its 'finite unwindings,' and inferences about its relation to other inductive definitions.

The first inference determines the direct generalizations of the inductive definition; an inductive definition is given the generalization

$$anchor \vee (test \wedge Domain(l_1) \wedge \cdots \wedge Domain(l_i))$$

which specifies the space from which the inductive definition is taken. Anything satisfying the inductive definition must be either the type of the anchor or amenable to recursive consideration through through the 'link functions' $l_1, l_2, \ldots, l_i$.

The second set of inferences is more complicated. An inductive definition has a potentially infinite set of specializations; we can choose to 'unwind' a given inductive definition to any extent and each finite unwinding is subsumed by the inductive definition. We cannot generate all these specializations but we would like to place any existing or newly constructed finite unwindings beneath the inductive definition in the lattice. As with the other types, we would like the inferences about recursive types to be lazy; only when a new finite winding is created do we make a subsumption inference.

To see how this is done in TYPICAL, consider the point at which a finite unwinding of an inductive definition T is created: a type is defined which is the intersection of the recursive type's test type with a set of image constraints (one for each link) into other finite unwindings of the type. If we assume inductively that all finite unwindings are already underneath the recursive type, the newly defined unwinding is beneath both the test type and the set of image constraints:

$$\boxed{\xrightarrow{l_1} T}, \boxed{\xrightarrow{l_1} T} \ldots \boxed{\xrightarrow{l_i} T}$$

Since it is underneath all of these, it is underneath their intersection:

$$test \wedge \boxed{\xrightarrow{l_1} T} \wedge \boxed{\xrightarrow{l_2} T} \wedge \cdots \wedge \boxed{\xrightarrow{l_i} T}$$

If we declare this intersection a specialization of T, all newly constructed finite unwindings of T will be beneath T.

Our inductive assumption was that all finite unwindings of T were actually beneath T already. To ensure this, we have to search for already-defined unwindings when T is created. If we only defined unwindings of T after T's definition, this would not be a problem; the single unwinding of T's anchor could be established as a definition. However, we don't have this assurance, so we have to search.

The search is fairly simple; we know that a type is a finite unwinding if it is beneath the test type and has constraints for each of the links which are finite unwindings. To find these, we search the lattice beneath the test type for types satisfying this description. These discovered types are then placed beneath the recursive type; once these are placed and the intersection above define, any subsequently defined unwinding will find the recursive type as a generalization.

A final set of inferences made by the inductive definition combinator connects new inductive definitions to other defined inductive definitions. An inductive definition is a specialization of another if it either has tests or anchors which are specializations of the other type's test or anchors, or if its list of links contains the links for the other type. The current implementation of this simply searches all the other defined inductive definitions to find types which satisfy these criterion.

This empty page was substituted for a blank page in the original document.

Chapter 5
Application: Control in **Cyrano**

TYPICAL was designed to represent the concepts and definitions manipulated and generated by the discovery program **Cyrano**. As TYPICAL was implemented and used, it became obvious that the many of the control distinctions made by **Cyrano** could be effectively represented by TYPICAL types. This chapter describes TYPICAL's application to control in the **Cyrano** program.

Control is mediated in **Cyrano** through the **indexing** of objects. Indexing finds the types which an object satisfies and then executes **daemons** attached to those types. The order in which these daemons are executed is determined by subsumption in the lattice; if a type s is below a type g in the lattice, the daemons for s are always executed before the daemons for g. Figure 5-1 shows the execution paths which might be taken through a lattice of types for the object '3'.

Indexing is similar to the mechanism of *realization* in the representation language KL-ONE [BS85] and its descendants. In realization, a description is located in a lattice of concept definitions; indexing extends this with a procedural execution component based on the object's location in the lattice.

Daemons and indexing submit to a variety of metaphors. We can consider the daemons as production rules, where the condition part of the rule is represented by a type in the lattice and the action component is the procedural implementation of the daemon. These production rules are ordered by the specificity of their conditions; subsumption in the lattice determines the order of 'rule' application.

Alternatively, if we imagine the daemons as 'statements' about objects, we can consider those statements as scoped over a set determined by the daemon's type. In this case, if the daemons are all consistent statements, order of daemon execution shouldn't matter.

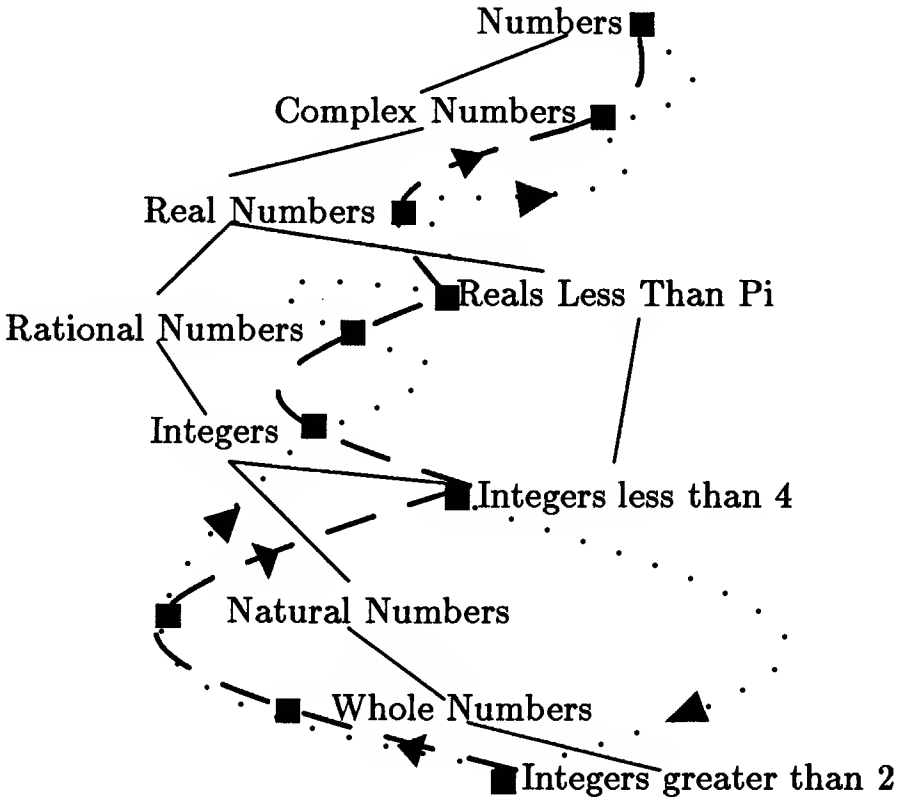

Figure 5-1. When indexing an object, we run the daemons up the lattice in such a way that no type has it's daemons run before any of its specializations. This is a potential sublattice of types satisfied by the number '3'; the dashed and dotted lines show two paths indexing might take through the lattice.

Another way to think about indexing is as a sort of perception; when an object is indexed, it is recognized and this recognition triggers some set of simple processes. The use of indexing as a control structure in **Cyrano** is driven by a view of most mental activity as just this sort of perception. Problem solving and knowledge accquisition, in this view, begin with perception and from perception the choice of action, inference, or construction is simple and immediate. We see a problem as an x-problem and immediately three methods for dealing with x-problems spring to mind. In most cases, any of the methods will work; in some, an attempt will reveal that the problem is actually an x_1-problem for which a single — always effective — method is known. This perspective is in part philosophical, part psychological, and part computational.

This chapter describes the implementation of indexing used in **Cyrano**. It begins by describing how indexing is used in **Cyrano** and then details its actual implementation. Since indexing is part of **Cyrano**'s inner loop, the efficency of its implementation is important; the final sections of the chapter describes several optimizations used in **Cyrano**'s

implementation.

5.1 Concept Generation in Cyrano

The definition of new concepts by **Cyrano** is driven and guided by the discovery of the properties or regularities of existing concepts. Certain regularities suggest certain definitions; for instance, if an operation (an implemented mapping between types of objects) is known to be a function (i.e. there is a unique $f(x)$ for each x), we can use the function to define a relation $\overset{f}{\asymp}$:

$$a \overset{f}{\asymp} b \longleftrightarrow a = f(b)$$

about which we might try and determine various properties. In **Cyrano** the process of defining such new concepts is implemented by daemons attached to types of concepts. For instance, we could define the procedure `OPERATION->RELATION`:

```
(define (operation->relation operation)
  (simple-type (lambda (pair)
                 (equal? (car pair) (operation (cadr pair))))
               (cross-product (range operation) (domain operation))))
```

This produces a subtype of pairs which corresponds to the relation which the operation determines. We could then define a daemon which calls this procedure on all operations known to be functional:[9]

```
(add-daemon! operation->relation functional-operations)
```

The `ADD-DAEMON!` procedure adds a daemon procedure to a particular type. Whenever an instance of that type is indexed, the procedure is called. When an operation is discovered to be functional, indexing the operation will fire the `OPERATION->RELATION` daemon.

However, the procedure `OPERATION->RELATION` only defines a new type; it doesn't do anything special with it. We could assume that other processes look at all new types and process them in some fashion. Alternatively, we could also simply reinvoke the indexer on the new definition. To specify this, we can define a higher order procedure `CONCEPT-GENERATOR`:

```
(define (concept-generator generator-function)
  (lambda (x) (index (generator-function x))))
```

which indexes the result of the generator function. Our call to `ADD-DAEMON!` now looks like this:

```
(add-daemon! (concept-generator operation->relation)
             functional-operations)
```

[9] In the actual implementation of **Cyrano**, 'operations' are simply relations whose instances are only determined by empirical generation; such a relation never *fails* to hold since there are no guarantees about what might eventually be generated. When such an operation or relation is found to be empirically functional, a new relation is defined which can be determined false, based on the recognized determinism of the operation.

We can use the indexing of new concepts to drive further concept definition; for instance, if we see a new relation, we can create a set of 'clusters' about that relation so that for any object subject to the relation we can define a type for objects related to it.

```
(define (left-cluster-function relation)
  (lambda (around-x)
    (simple-type
      ;; This is the predicate for the cluster:
      (lambda (y) (in? (list around-x y) relation))
      ;; This is the type it specializes; the right hand side (CADR)
      ;; constraints on the type.
      (mapping-constraint car (mapping-constraint cdr relation)))))
```

This creates a new concept generator which produces types from objects. This generator uses the MAPPING-CONSTRAINT procedure (Section 4.3.1; Page 30) to determine the space over which the relation is defined. We can use this function as a concept generator: the concepts it produces are procedures which happen to be new concept generators.

```
(add-daemon! (concept-generator left-cluster-function)
             relations)
```

The generated clustering function is indexed when it is created and this indexing may lead to further definitions or other activities. For instance, we could define a daemon which would take indexed procedures like clustering functions and install them as daemons on the appropriate types:

```
(add-daemon! (lambda (x) (add-daemon! x (function-domain x)))
             (<AND> (image-constraint function-domain lisp-objects)
                    (image-constraint function-range  lisp-objects)))
```

The clustering function maps from a space of objects into a space of types. This daemon takes any such function and establishes *it* as a daemon.

5.1.1 Controlling Concept Generation: Foci and Potential Foci

The concept generation mechanism described above is explosive, since defining a new concept of one sort may lead to the definition of a new concept of another sort which may lead to the definition of a new concept of the first sort, and so forth. For instance, an operation might lead to a relation via OPERATION->RELATION which might lead to a new operation via LEFT-CLUSTER-FUNCTION which might lead to a new relation via OPERATION->RELATION and so forth. **Cyrano** attempts to control this potential explosion by defining two special classes of definitions: **foci** and **potential foci**.

Foci are objects which are known to be interesting: by empirical experimentation, user assertion, or particular connection with other foci. Potential foci are objects which have been created and *might* be interesting. New concept definitions spring *only* from foci; potential foci become foci only when they exhibit interesting regularities. Figure 5-2

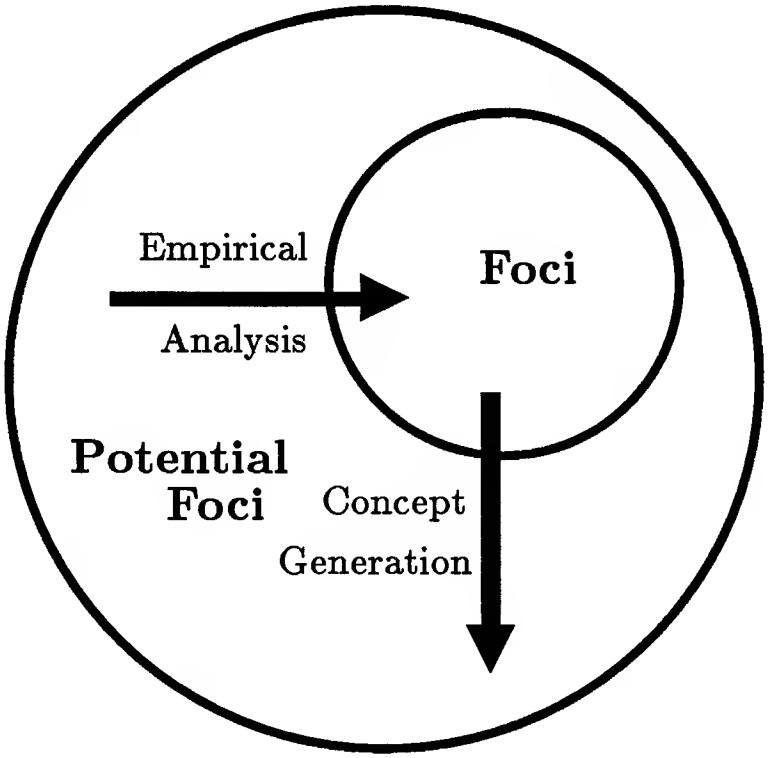

Figure 5-2. Cyrano's control structure divides objects into two classes: foci and potential foci. New concept definitions spring *only* from foci and potential foci become foci only when they exhibit interesting regularities.

illustrates this process. Foci and potential foci are empirical types; unless an object is declared as a foci or potential foci, its status is undetermined.

```
;;; Potential foci are an empirical subtype of lisp objects.
(define potential-foci (empirical-type lisp-objects))
;;; Foci are an empirical subtype of potential foci.
(define foci (empirical-type potential-foci))
```

We redefine CONCEPT-GENERATOR to add its results to Potential Foci:

```
(define (concept-generator generator-function)
  (lambda (x)
    (let ((new-concept (generator-function x)))
      (put-in-collection! new-concept potential-foci)
      (index new-concept))))
```

and define an operator on types which returns the foci subtype for a given type:

```
(define (foci-type base-type) (<AND> foci base-type))
```

And then our ADD-DAEMON! forms would look like:

```
(add-daemon! (concept-generator operation->relation)
             (foci-type functional-operations))
```

Of course, it might well be that most functional operations are interesting for their very deterministic nature. In this case (where functional operations are special sort of foci), the daemon would just be added to [Functional Operations] since <AND> applied to [Foci] and [Functional Operations] just returns [Functional Operations].

But if this were the case, we still might want to restrict the use of the generator OPERATION->RELATION. For instance, if some operations were extremely expensive, converting them into a relation type in the lattice would bring about their constant invocation in checking random pairs for satisfaction of the relation. We might like to keep OPERATION->RELATION from being applied to expensive operations. The mechanism used in Cyrano for special cases like this one is called *inhibition.*

5.1.2 Inhibiting Daemon Execution

When an object is indexed, the lattice is ascended and daemons along the path executed. Cyrano's indexer provides a facility for *inhibiting* these daemons for particular objects; when an object is indexed, it's inhibited daemons are never executed. The procedure INHIBIT-DAEMON! inhibits a daemon for a particular object; for instance, we could inhibit relation construction for the operation SAVE-WORLD-STATE:

```
;;; Don't try converting world-state-saving into an operation.
(inhbit-daemon! (concept-generator operation->relation)
                save-world-state)
```

This allows individual instances to inhibit particular daemons. This is useful, but we can do better by defining daemons which inhibit other daemons. For instance:

```
(define (inhibitor for-daemon)
  (lambda (x) (inhibit-daemon! for-daemon x)))
```

allows us to inhibit OPERATION->RELATION for all instances of the type [Expensive Operations]:

```
;;; Never try converting expensive operations into relations.
(add-daemon! (inhibitor operation->relation)
             (foci-type (<AND> functional-operations
                               expensive-operations)))
```

The effectiveness of this inhibitor depends on the ordering of daemon execution by the indexer. It relies on the execution of daemons for in order of specificity; since the type with the inhibitor is below the type with the concept generator, the inhibitor will be executed before the concept generator is reached.

One variation on inhibition allows the inhibition of classes of daemons (assuming a taxonomy of daemon types); rather than simply inhibiting a particular daemon for an object, we inhibit classes of daemons for the object. This was tried in an earlier version of Cyrano, but abandoned as it became clear that it was never being used and that the more specific case of inhibiting a single daemon was all that was neccessary.

5.2 Implementing INDEX

To introduce the implementation of indexing, we begin by assuming a procedure MAPTYPES which we will eventually define. MAPTYPES takes a procedure and an object and calls the procedure on each type the object satisfies, in subsumption order. Thus if a type s is below a type g, the procedure is called on s before g.

A version of indexing without inhibition might look like this:

```
(define (index x)
  (define (run-daemon daemon) (daemon x))
  (define (run-daemons type)
    (for-each run-daemon (daemons-for-type type)))
  (maptypes run-daemons x)
  x)
```

where the procedure DAEMONS-FOR-TYPE extracts a list of procedures from the incidental properties of the type. Given this, ADD-DAEMON! can be defined thus:

```
(define (add-daemon! daemon type)
  ((modifier daemons-for-type) type
   (cons daemon (daemons-for-type type))))
```

which calls the MODIFIER of DAEMONS-FOR-TYPE to add a new daemon procedure.

We can maintain an inhibitions store by a procedure GET-INHIBITIONS-STORE; called on a single parameter, GET-INHIBITIONS-STORE returns a cons cell whose CDR is a list of inhibited daemons for x. INHIBIT-DAEMON! will look like this:

```
(define (inhibit-daemon! daemon for-object)
  (let ((inhibitions-store (get-inhibitions-store for-object)))
    (set-cdr! inhibitions-store
              (cons daemon (cdr inhibitions-store)))))
```

and we will update INDEX to check this store:

```
(define (index x)
  (let ((inhibitions (get-inhibitions-store x)))
    (define (run-daemon daemon)
      (if (not (member daemon (cdr inhibitions)))
          (daemon x)))
    (define (run-daemons type)
      (for-each run-daemon (daemons-for-type type)))
    (maptypes run-daemons x)
    x))
```

This version of INDEX passes over inhibited daemons and also allows executing daemons to modify the inhibitions store in a visible way. This version of INDEX can be spiced up with facilities for tracing daemon execution, keeping timing statistics for daemons, etc.

Much of the work of INDEX is obviously done by the procedure MAPTYPES; the implementation of MAPTYPES is examined in the next section.

```
(define (maptypes procedure object)
  (let ((visited-nodes ()))
    (define (maptypes-under type)
      (cond ((member type visited-nodes)
             ;; Don't visit nodes you've already checked.
             #F)
            ((in? object type)
             ;; If the node may have satisfied children,
             ;; Mark the node as not to be revisited
             (set! visited-nodes (cons type visited-nodes))
             ;; Visit its children,
             (for-each maptypes-under (specializations type))
             ;; And THEN call the procedure on it.
             (procedure type))
            (ELSE #F)))
    (maptypes-under lattice-top)))
```

Figure 5-3. This implementation of MAPTYPES does a depth first descent of the lattice, calling procedure each time it finishes with the children of a node and keeping a track of visited nodes so that it doesn't call procedure (or expand children) twice on the same node.

5.2.1 Implementing MAPTYPES

The MAPTYPES procedure takes a procedure and an object and applies the procedure to all of the types the object satisfies. A rough-cut implementation of MAPTYPES is shown in Figure 5-3. It does a depth first descent of the lattice, calling procedure on each type it encounters which is satisfied by object. It prunes the specializations of those types which don't satisfy the object. If a type is satisfied, its generalizations are satisfied; by contraposition, none of the specializations of an unsatisfied type can be satisfied. This procedure also keeps track of the nodes it has visited on a list; thus it will never call procedure on any node twice.

The order in which procedure is called is the *topological sort* of the sublattice of types satisfied by the object. We know that when we call procedure on a type, we have already called procedure on all the types below it. Thus, we will never call procedure on a type unless we have already called procedure on all the types below it.

Two pragramatic problems emerge with this simple implementation. First, the list of visited nodes may get quite long, adding a linear factor to the time a given application will take. Furthermore, the calling stack is likely to get quite deep. By writing MAPTYPES iteratively, we can avoid the second problem and by keeping a set of *visit marks* (in a bit-string) we can finesse the $O(n)$ lookup on visited nodes (we assume a constant time bit

```
(define (maptypes procedure object)
  (let ((visit-marks (make-empty-bit-string)))
    ;; We use a bit string, indexed by type identifiers, to keep visit marks
    (define (topological-map stack)
      ;; Each element of the stack is either a type or a list of a single type.
      ;; Types are nodes to be tested and (potentially) expanded; the list of
      ;;  a single type indicates a type which has already been expanded.
      (if (null? stack) () ; If the stack is empty, we are done.
          (cond ((list? (car stack))
                 ;; If we have returned to a node we expanded,
                 ;; call procedure, set the visit mark, and iterate.
                 (procedure (car (car stack)))
                 (set-bit! visit-marks (type-id (car stack)))
                 (topological-map (cdr stack)))
                ((in? object (car stack))
                 ;; Otherwise, if the node/type is satisfied, push it on the
                 ;; stack as `already-expanded' and expand it.
                 (push-specializations
                   (specializations (car stack))
                   (cons (list (car stack)) (cdr stack))))
                ;; Otherwise, keep going down the stack.
                (ELSE (topological-map (cdr stack))))))
    (define (push-specializations specializations stack)
      ;; This adds a list of specializations to the search stack.
      (if (null? specializations)
          (topological-map stack) ; If there is nothing more to push, continue.
          (if (check-bit visit-marks (type-id (car specializations)))
              ;; If the specialization has been visited already, don't bother
              ;;  visiting it again.
              (push-specializations (cdr specializations) stack)
              ;; If the specialization is new, add it to the stack and keep pushing.
              (push-specializations (cdr specializations)
                 (cons (car specializations) stack)))))
    (topological-map (list lattice-top))))
```

Figure 5-4. This iterative version of the MAPTYPES procedure keeps track of where it has been with visit marks.

string reference operation). Figure 5-4 shows this implementation. Though a bit involved, it can execute tail-recursively consing only the internal stack argument stack.

Another problem with both the iterative and recursive implementations emerges if we consider the cost of the call to IN?. MAPTYPES calls IN? $O(n)$ times (where n is the size of

the lattice); at each point IN? calls a type predicate to determine satisfaction for object. If the type predicate is an analytic combination of other analytic types, the worst-case cost (in calls to primitive type predicates) for a call to IN? is also $O(n)$, giving and $O(n^2)$ bound for a given indexing operation. Indexing is a common operation for Cyrano and the next section describes how this $O(n^2)$ time bound is practically improved by using a cache for type satisfaction.

5.2.2 Optimizing MAPTYPES: Satisfaction Caches

Cyrano calls the indexer on every new concept and example; it is part of the program's inner loop. We would like to improve on the $O(n^2)$ time bound for indexing and we can do so by using a *satisfaction cache.* We introduce a procedure WITH-SATISFACTION-CACHE which takes an object parameter and a zero-argument procedure and calls the procedure in an environment where satisfaction information about the object is cached. Figure 5-5 (Pag 46) eimplements a version of the satisfaction cache. It works by dynamically redefining the TYPICAL procedure SATISFIES? to use a pair of bit strings as a cache for satisfaction information.

If the MAPTYPES procedure of Figure 5-3 were renamed MAPTYPES-1, we could define a more efficent version:

```
(define (maptypes procedure object)
  (with-satisfaction-cache object (maptypes-1 procedure object)))
```

which will involve only $O(n)$ basic predicate calls.

In the actual implementation in Cyrano, the details differ, but the general idea of fluidly rebinding the satisfaction function makes indexing significantly cheaper. One useful extension in the actual implementation is to maintain type caches for objects *between* calls to MAPTYPES. This speeds up repeated indexings by a significant factor, since most 'basic' predications only need to happen once. It does however, engender a host of problems with cache invalidation. The partial solution currently used in Cyrano is to reset the stored cache explicitly when cached satisfaction information is known to be invalid.

5.3 Summary

In this chapter I described how *indexing* is used in the control structure of the Cyrano program. Indexing takes an object and locates it in the lattice of types; this location is then used to find *daemon procedures* to call on the object. These daemon procedures are attached to individual types and if an object satisfies a type, its corresponding daemons are called. Daemons are executed in order of specificity; if a type s is below a type g, the daemons for s are called before the daemons for g.

Daemon execution applies a procedure to a topological sort of the sublattice of types satisfying an object. This operation is made more efficient by the use of a *satisfaction cache* to store satisfaciton information in the lattice.

```
(define (with-satisfaction-cache for-object procedure)
  ;; We use two bit strings, referenced by type indices, to cache
  ;;  satisfaction information.
  (let (;; Determines if the cache accurately reflects satisfaction of a type.
        (cached?-vector (make-empty-bit-string))
        ;; Determines if a type is actually satisfied.
        (satisfies?-vector (make-empty-bit-string))
        ;; The default IN? procedure.
        (outer-satisfies? satisfies?))
    (define (inner-satisfies? object type)
      (cond ;; If the call to SATISFIES? is not for-object, pass it on:
            ((not (eq? object for-object))
             (outer-satisfies? object type))
            ;; If the satisfaction information is cached, return the cached value:
            ((check-bit cached?-vector (type-id type))
             (check-bit satisfies?-vector (type-id type)))
            ;; Otherwise go ahead and compute satisfaction:
            (ELSE (let ((satisfied? (outer-satisfies? object type)))
                    (cond ((and (defined? satisfied?) satisfied?)
                           ;; If the type is really satisfied (defined and true),
                           ;;  set the appropriate bit in both the cache vector and
                           ;;  the satisfaction vector.
                           (set-bit! cached?-vector    (type-id type))
                           (set-bit! satisfies?-vector (type-id type)))
                          ((defined? satisfied?)
                           ;; If the type is really unsatisfied (defined and false),
                           ;;  set the appropriate bit in the cache vector but leave
                           ;;  the satisfaction vector clear.
                           (set-bit! cached?-vector    (type-id type))))
                    ;; Finally, return the result of the satisfaction query.
                    satisfied?))))
    (fluid-let ((satisfies? inner-satisfies?)) (procedure))))
```

Figure 5-5. The procedure WITH-SATISFACTION-CACHE fluidly binds TYPICAL's SATISFIES? procedure to a procedure which keeps track of satisfaction information for an object in a pair of bit strings.

This empty page was substituted for a blank page in the original document.

Chapter 6
Application: Hypotheses in Cyrano

As a language for constructing definitions, TYPICAL can be viewed as a deductive inference engine for **Cyrano**; it draws the neccessary consequences of a given definition and makes these consequences visible to the program as structure within the lattice. TYPICAL was at first designed for largely this role: a language for representing and making inferences about concepts defined by a vocabulary of combinators.

In the last chapter, I described how TYPICAL is used in the control structure of the **Cyrano** program, implementing a heuristic rule engine where applicability conditions are specified by types in the lattice. TYPICAL's lattice inferences provided a superstructure for indicating that certain rules had priority over others.

In this chapter I describe how TYPICAL is used to support **Cyrano**'s inductive inferences. Just as the deductive capacities of TYPICAL support the heuristic inference mechanism described in the preceding chapter, the confirmation or disconfirmation of empirical hypotheses is also constructed on the representational substrate of TYPICAL's definition and inference capabilities. In particular, TYPICAL types are used to represent examples and counterexamples of emprical regularities.

Confirmation of empirical properties by TYPICAL uses the indexing mechanism presented in the previous chapter. The confirmation process defines two disjoint types in TYPICAL's lattice: a type for counterexamples and a type for examples. Index daemons attached to these types then mediate the confirmation process by reacting to the indexing of counterexamples or examples. When a counterexample is indexed, the property is disconfirmed. When a sufficent number of examples is indexed, the property is tentatively confirmed. The definition of these classes and (in part) the generation of possible candidates for them is the focus of this chapter.

6.1 Sample and Evidence Types

When **Cyrano** indexes a particular type definition, it naturally falls into various 'meta types' in TYPICAL's lattice. Attached to these types are daemons which propose empirical properties the definition might satisfy; these hypotheses initiate the creation of 'experiments' which seek to confirm or disconfirm the proposed empirical property. Of course, such confirmations — like the hypotheses which lead to them — are neccessarily heuristic; the confirmation mechanism remains ever on the lookout for counterexamples to empirically confirmed ('so far') properties.

Empirical properties are 'accidental'; they are things which are true about types not in virtue of their definition, but in virtue of the world of objects they distinguish. Since **Cyrano**'s knowledge of the world is largely expressed in terms of subsumption of definitions in the lattice, empirical regularities are expressed as accidental subsumption relations in the lattice. Every regularity is eventually expressed by the suprising subsumption of one type beneath another; each recognized empirical property represents the suprising containment of one class of objects within another.

The accidental subsumption which corresponds to a potential regularity is determined by an *empirical class* for the particular sort of regularity. Empirical classes are implemented in TYPICAL by *divided collections* (Section 4.4.2, Page 32). Each empirical class specifies two attached functions: a sample space generator and an evidence space generator. To confirm a regularity for a particular instance, these functions are called on the instance, returning (respectively) a sample type and an evidence type. The regularity is satisfied if the sample type is subsumed — either neccesarily or empirically — under the evidence type.

Confirming a property is a matter of demonstrating either the statistical plausibility or strict impossibility of this accidental subsumption. Statistical plausibility is indicated by a 'convincing number' of instances of the sample type which also satisfy the evidence type; strict impossibility is indicated by an instance of the sample type which is not an instance of the evidence type.

The simplest empirical classes have sample space generators which are the identity relation:

$$R_{Sample}(x) = x$$

and evidence functions which are some constant pre-existing type:

$$R_{Evidence} = C_R.$$

Such classes express regularities like 'All X's are red,' 'r is an identity relation,' or 'f is monotonically increasing'; they are statements that membership in the class implies membership in some larger class.

More complicated test spaces combine or transform the concept being analyzed. For instance, relations in **Cyrano** are represented as types of pairs (e.g. EQUAL is represented as the type of all pairs of equal lists). The empirical class of symmetric relations has a

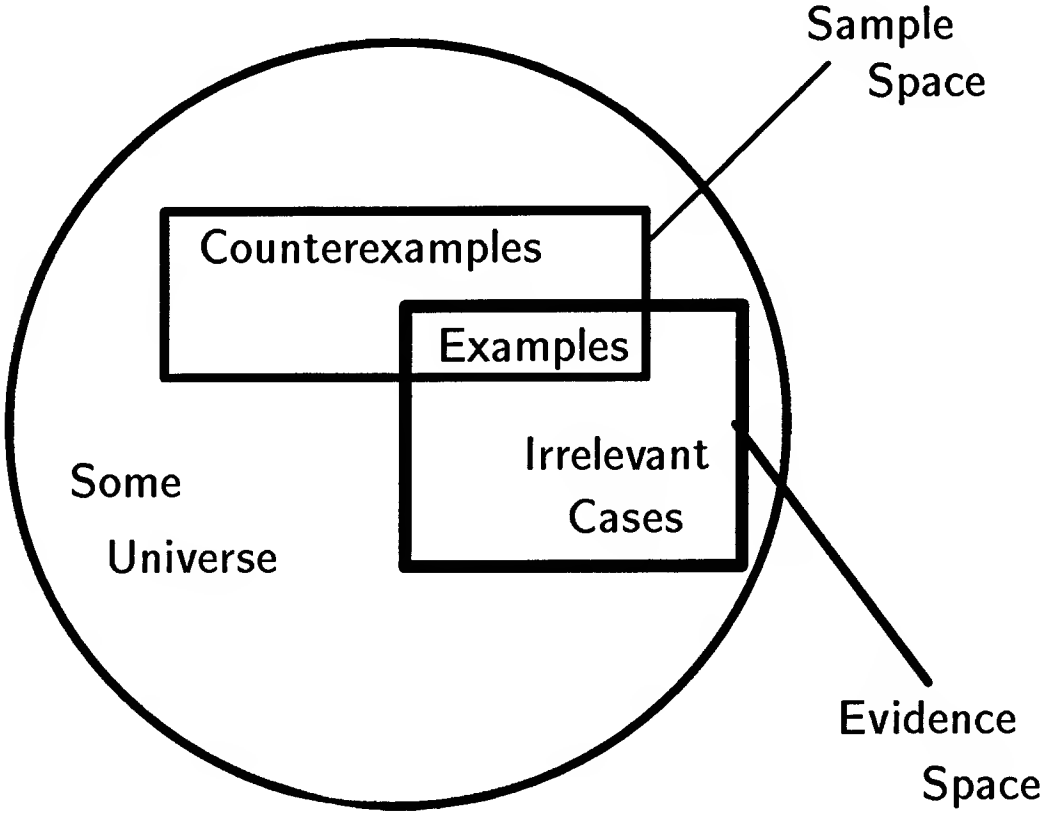

Figure 6-1. Empirical classes describe empirical regularities by potential accidental subsumption/subset relations in the lattice.

test function which generates the inverse of a relation and a confirmation space which is the relation itself. We can define an inverse of a relation R by using the mapping

$$twister(\langle x, y\rangle) \equiv \langle y, x\rangle$$

in an image constraint to define $\boxed{\overset{twister}{\longrightarrow} R}$. This is the inverse of R which we can then use to determine if R is symmetric:

$$Summetric_{Sample}(R) = \boxed{\overset{twister}{\longrightarrow} R}$$
$$Symmetric_{Evidence}(R) = R$$

If $Symmetric_{Sample}(R)$ is beneath $Symmetric_{Evidence}(R)$, then R is symmetric.

The process of confirmation defines two new types from the sample evidence spaces: a type for examples and a type for counterexamples. Logically, for a regularity K, the set of counterexamples to x satisfying K is

$$K_{Sample}(x) \wedge \neg K_{Evidence}(x)$$

and the set of positive examples for x satisfying K is the intersection

$$K_{Sample}(x) \wedge K_{Evidence}(x)$$

Figure 6-1 illustrates the specification of these types in terms of overlapping regions. Instances in the class of counterexamples instantly disconfirm the regularity in question. Instances in the class of examples have no such immediate effect. A positive conclusion must wait for some convincing corpus of examples, and even then such a conclusion is at most tentative. Even if every preceding sentence of a paragraph ended with a period, that makes no guarantees about this one!

Counterexamples and examples are noticed by Cyrano through index daemons attached to the classes of examples and counterexamples. When a potential member of the sample space is generated, it is indexed in the lattice. If the potential member is really in the sample space, the object will fall into either the class of examples or the class of counterexamples, triggering the daemons attached to each of these classes. If the counterexample daemon is triggered, it disconfirms the regularity and removes both itself the example collecting daemon. If the example daemon is triggered, it checks if there is already a sufficent quota of examples and — if there is — tentatively confirms the regularity. Upon confirming a regularity, the examples daemon removes itself from activity, but leaves the counterexample daemon untouched. In this way, eagerly made mis-confirmations can be caught when later counterexamples show up. The exact manner in which such after-the-fact fixes should work remains an open problem.

6.2 Implementing Confirmation

Confirmation combines TYPICAL's definition facilities and the indexing facilities implemented in Chapter 5. In the following section, we will present a version of Cyrano's confirmation implemenation. We assume the indexing implementation of the last chapter and an additional procedure, `GENERATE-EXAMPLES`, which sets up machinery for generating and indexing examples of a type.

Confirmation involves the definition of types for examples and counterexamples, and the attachement of appropriate daemons to these types. The definition of these types derives from the sample space/type and evidence space/type functions of some empirical regularity; defining these functions and the empirical classes they define constiutes the first part of the implementation of confirmation.

Empirical classes are created by the `EMPIRICAL-CLASS` procedure. This procedure calls the `DIVIDED-COLLECTION` combinator, described in Section 4.4.2 (Page 32). `EMPIRICAL-CLASS` will take three arguments: a type, a sample-space generator, and an evidence-space generator. procedure. The first argument specifies the generalization of the generated empirical class; the sample-space and evidence-space generators take an object and return the sample

```
(define (hypothesis subject property)
  (let ((sample ((sample-space-generator property) subject))
        (evidence ((confirmation-space-generator property) subject)))
    (cond ((subsumed-by? sample evidence)
           (message $NL "!!! By definition, " subject " is in " property)
           (message $NL "!!! Instances of " sample " are always in " evidence)
           (assert! subject property))
          ((disjoint? sample evidence)
           (message $NL "!!! By definition, " subject " cannot be in " property)
           (message $NL "!!! Instances of " sample " are never in " evidence)
           (assert! subject (complement property)))
          (ELSE (setup-counterexamples-daemon subject property sample evidence)
                (setup-examples-daemon subject property sample evidence)
                (generate-examples sample)))))
```

Figure 6-2. The HYPOTHESIS procedure generates sample and evidence spaces and sets up an experiment if neccessary.

and evidence types for the regularity:

```
(define (empirical-class beneath
                         sample-space-generator evidence-space-generator)
  (let ((property (divided-collection beneath)))
    ((modifier sample-space-generator)
       property sample-space-generator)
    ((modifier evidence-space-generator)
       property evidence-space-generator)
    property))
```

EMPIRICAL-CLASS generates a dividied collection and annotates it with sample and evidence space generators. These annotations are added by using the higher-order 'modifier' procedure which returns a procedure for modifying a property.

Confirmation is initiated by the HYPOTHESIS procedure which generates sample and evidence spaces for a property and — if neccessary — sets up the experimental apparatus to try and confirm the property. A possible implementation of HYPOTHESIS is shown in Figure 6-2. This takes two arguments: an object an an empirical property. Fetching the sample and evidence space generators from the property, it applies these to the subject to generate the appropriate sample and evidence spaces.

Before beginning an actual experiment, it checks that the regularity is not 'trivially' confirmed; if the defined types are already beneath one and other or known to be disjoint, no data is needed to confirm or disconfirm the empirical property. This subsumption and disjointness information used here is generated by TYPICAL in the process of type combination and inference initiated by the construction of sample and evidence types. In

```
(define (setup-counterexamples-daemon subject property sample evidence)
  (let* ((counterexamples (<AND> sample (complement evidence))))
    ;; This is the actual counterexample daemon.
    (define (notice-counterexample x)
      ;; If you find a counterexample, announce it:
      (message $NL "!!! Found a counterexample excluding "
               subject " from " property)
      (message $NL "!!! Declaring " property " unsatisfied for " subject)
      ;; Remove the apparatus for noticing counterexamples.
      (remove-daemon! counterexamples notice-counterexample)
      ;; And assert its membership in the appropriate empirical class.
      (put-in-collection! subject (complement property))
      ;; And (re)index the newly declared object.
      (index subject))
    ;; Add the counterexample daemon.
    (add-daemon! notice-counterexample counterexamples)))
```

Figure 6-3. The procedure SETUP-COUNTEREXAMPLES-DAEMON defines an internal procedure which notices counterexamples to a proposed empirical property.

providing subsumption and disjointness information, TYPICAL is serving as an *inference engine* for 'proving' properties from the definition of sample and evidence spaces.

If TYPICAL's inferences do not immediately confirm of disconfirm the property, HYPOTHESIS sets up the daemons which notice examples or counterexamples and then begins a process of actually generating potential samples to be noticed.

Figure 6-3 shows a possible implementation of SETUP-COUNTEREXAMPLES-DAEMON. It defines a counterexample space by intersecting the sample space with the complement of the evidence space. It then defines an internal procedure NOTICE-COUNTEREXAMPLE to use as a daemon. This procedure:

1. Announces the presence of a counterexample to the user.
2. Removes itself from the counterexamples type, so that it will not needlessly fire again.
3. Adds the potential instance of the empirical regularity to the complement of the regularity's empirical class.
4. Re-indexes the once-potential instance to allow new daemons to fire based on its new classification.

The NOTICE-COUNTEREXAMPLE procedure is made a daemon on the type of counterexamples by the ADD-DAEMON! procedure of the indexing implementation.

Figure 6-4 shows a possible implementation of SETUP-EXAMPLES-DAEMON. It defines the space of examples by intersecting the sample and evidence spaces it is given. It then defines an internal procedure NOTICE-EXAMPLE to use as a daemon on this example space.

```
(define (setup-examples-daemon subject property sample evidence)
  (let ((examples (<AND> sample evidence))
        (examples-seen ()))
    ;; The THRESHOLD is the number of examples required for confirmation.
    (define (notice-example x)
      (if (not (member x examples-seen))
          (set! examples-seen (cons x examples-seen)))
      (if (defined? (has-type? subject property))
          (remove-daemon! examples notice-example)
          (if (> (length examples-seen) examples-threshold)
              ;; If there are `enough' examples, announce your discovery
              (begin
                (message $NL "!!! Found " (length examples-seen) "examples"
                         " of " property " for " subject)
                (message $NL "!!! Declaring " property
                         " tentatively satisfied for " subject)
                ;; Assert it into the appropriate empirical class,
                (put-in-collection! subject property)
                ;; and (re)index it based on this new knowledge.
                (index subject)))))
    (add-daemon! notice-example examples)
    notice-example))
```

Figure 6-4. The procedure SETUP-EXAMPLES-DAEMON defines an internal procedure which notices examples supporting a proposed empirical property.

NOTICE-EXAMPLE keeps a single piece of state between invocations: a list of examples it has seen. If it is called on an example which has already been seen, it does nothing; if the example is new, it adds it to the list of known examples and checks the length of this list. If the length is past some ad-hoc threshold, it tentatively asserts the relation satisfied by:

1. Announcing that the threshold has been passed and that the regularity is being tentatively asserted.
2. Adding the potential instance of the empirical regularity to the regularity's empirical class.
3. Re-indexing the potential instance to allow new daemons to fire based on its new classification.

An interesting wrinkle in the definition of NOTICE-EXAMPLE is that it is self-disabling; if the regularity it is attempting to illustrate has already been determined (positively or negatively) when it is called, it removes itself as a daemon from the type of examples. Note that we do not remove the counterexamples daemon if the regularity is assumed satisfied; we still want to be able to be proven wrong.

As with the counterexamples daemon, the procedure ADD-DAEMON! adds the procedure NOTICE-EXAMPLE to the type of examples.

6.3 Confirming Cliches

Cyrano us the confirmation mechanism described above to recognize a variety of complex regularities in the domains presented to it; these regularities are organized into broad classes called *cliches* after Chapman's usage [Cha83] [Cha86]. Cliches are highly exploitable, domain independent, formally specified properties of representations; not a theory of representation themselves, they are properties of domains and the representations of domains. Central to the theory of cliches is the thesis that most understanding is structured around a small set (less than 1000) of powerful ideas which are used from domain to domain. Examples of cliches are notions of continuity, ordering, partitions, equivalence classes, or symmetry. The important properties of cliches are: their formal specification (given a representation and represented examples, it is easy to tell if a cliche is present); their sparseness (it is estimated that there are less than a thousand general purpose cliches); and their domain independence (a single cliche, like continuity, will find a place in mathematics (of course), physical reasoning, action planning, etc). Cliches are similar to Minsky's notions of 'concept germs' as presented in [Min86]. Methodologically, the general notion of cliches and cliche theory arose from a generalization of *program cliches* presented in [RS76].

Cyrano structures its experimental activities around identifying these domain independent cliches. This section presents the implementation of a handful of these regularities using confirmation mechanism described above. The regularities describe properties of relations, functions, and mappings represented by types in TYPICAL's lattice.

In particular, these objects are defined as particular types of *pairs* in the lattice; a relation, mapping, or function is a special type of pair, specifying a subset of all possible pairs. To begin, we define the class of pairs and operations on them; all such objects are beneath the type of pairs in general, and have right and left elements:

```
;;; This is the supertype of all types of pairs.
(define pairs (cross-product lattice-top lattice-top))

;;; We define synonyms LEFT and RIGHT for CAR
;;; and CADR. Since these are already declared as mappings
;;; by TYPICAL, it is unneccessary to declare them anew.
(define left car)
(define right cadr)
```

Given the class of pairs, we define the class of pairings; all relations, functions, or mappings will be beneath this class which is the power set of pairs:

```
(define pairings (power-set pairs))
```

For any specialization of Pairings, we might wish to individually extract the left and right constraints which the type places on its elements. To do this we use the procedure

MAPPING-CONSTRAINT (Section 4.3.1; Page 30) to define:

```
;;; This returns the constraints on the left side of a pairing.
(define (left-constraints type) (mapping-constraint car type))

;;; This returns the constraints on the right side of a pairing.
(define (right-constraints type)
  (mapping-constraint car (mapping-constraint cdr type)))
```

Given this framework for describing pairings, we can look for many regularities in these pairings. One regularity is whether a given relation is deterministic: whether a given right hand side is always associated with a given left hand side or vice versa. Other regularities are algebraic; is a relation "$\sim$" reflexive

$$x = y \longleftrightarrow x \sim y,$$

anti-reflexive

$$x = y \longleftrightarrow \neg(x \sim y),$$

symmetric

$$x \sim y \longleftrightarrow y \sim x,$$

or anti-symmetric

$$x \sim y \longleftrightarrow \neg(y \sim x).$$

6.4 Recognizing Determinism

When we speak of a pairing being deterministic, we mean that over all instances of the pairing, pairs with identical right or left sides have identical left or right sides. This is loosely related to the notion of determinism underlying causality; a deterministic causal law is one which claims that all X events are followed by Y events.

We define two sorts of determinism for a pairing: left deterministic pairings and right deterministic pairings. For a pairing P of elements $\langle x, y \rangle$, P is left deterministic if and only if $f(x) = y$ is a function; P is right deterministic if and only if $f(y) = x$ is a function. Closer to the language of pairs, if for any two elements of P, if left-hand identity implies right-hand identity, P is left deterministic; if right-hand identity implies left-hand identity, P is right deterministic.

To recognize these sorts of determinism in a pairing, we need to define sample and evidence spaces where containment of the sample space in the evidence space occurs only when the pairing is deterministic. Since determinism is a property of particular pairings relative to all other pairings, this requires a way to compare all the elements of a given pairing. We can do this by establishing a pairing of the pairings; this is a type satisfied by pairs whose first and second elements are both instances of the pairing we are testing. If $\boxed{R}$ is a pairing we are examining, the space we will be experimenting in is the cross product type $(\boxed{R} \times \boxed{R})$. The complete extension of this type is all the permutations of R (viewed as a set of pairs) with itself.

But this is not exactly the sample space we want; it is too large. Determinism says nothing about what happens when their is no identity of left or right sides. Right determinism makes a claim about pairs of pairs whose left-hand sides are equal; left determinism makes a claim about pairs of pairs whose right hand sides are equal. To begin defining test and confirmation classes (and functions), we need to define the class of left and right identical pairs, which we do by escaping to LISP to define the predicates RIGHT-EQUAL? and LEFT-EQUAL?: [10]

```
;;; First a predicate which makes sure that something is really a pair of pairs.
(define (pair-pair? x)
  (define (a-pair? y) (and (list? y) (= (length y) 2)))
  (and (a-pair? x) (a-pair? (first x)) (a-pair? (second x))))
;;; Returns true for pairs of pairs whose right hand elements are the same:
(define (right-equal? x)
  (and (pair-pair? x)
       (equal? (right (first x)) (right (second x)))))
;;; Returns true for pairs of pairs whose left hand elements are the same:
(define (left-equal?  x)
  (and (pair-pair? x)
       (equal? (left (first x)) (left (second x)))))
;;; And now we define the corresponding types for these predicates:
(define right-equals (simple-type right-equal? (list pair-pairs)))
(define left-equals (simple-type left-equal? (list pair-pairs)))
```

Given these characterizations of the test space, the confirmation space is simply the other side of the equality: those pair pairings which are right equal or left equal when the test space is left equal or right equal. So we can define the empirical class Right Deterministic as:

```
(define right-deterministic
  (empirical-class pairings
    (lambda (r) (<AND> (cross-product r r) right-equals))
    (lambda (r) (<AND> (cross-product r r) left-equals))))
```

A pairing R is right deterministic if pairs of Rs which are right identical are also left identical. The definition of left-deterministic pairings is symmetric to that of right-deterministic pairings:

```
(define left-deterministic
  (empirical-class pairings
    (lambda (r) (<AND> (cross-product r r) left-equals))
    (lambda (r) (<AND> (cross-product r r) right-equals))))
```

[10] Described predicates are, by convention, not allowed to signal type errors; if their argument or some component of it is not of the correct type, they simply return #F (false). TYPICAL's analytic combinators ensure this in the functions they generate, but simple types, are handed an opaque predicate for which they cannot guarantee this property. The user must ensure that the predicates handed to SIMPLE-TYPE (like RIGHT-EQUAL?) explicitly type their arguments.

Given the definitions for right and left deterministic pairings, we can define daemons which will set up 'experiments' for any pairings which are defined. As in Chapter 5, we prefer to define a daemon generator:

```
(define (hypothesize property) (lambda (x) (hypothesis x property))
```

and proceed to use this in adding daemons:

```
(add-daemon! pairings (hypothesize right-deterministic))
(add-daemon! pairings (hypothesize left-deterministic))
```

These daemons will fire on any pairings and set up both the experimental apparatus (sample space, evidence space, and sample generatoring machinery) and the confirmation and disconfirmation daemons for 'observing' the results of the experiment.

We can use membership in empirical classes — resulting from experiments like those above — to analytically determine membership in other classes. For instance, we can use the definitions of determinism above to define analytic intersections describing four classes of mappings (represented as pairs):

```
(define one-to-one
  (<AND> right-deterministic left-deterministic))
(define many-to-many
  (<AND> (complement left-deterministic)
         (complement right-deterministic)))
(define one-to-many
  (<AND> right-deterministic (complement left-deterministic)))
(define many-to-one
  (<AND> left-deterministic  (complement right-deterministic)))
```

Whenever a new relation is indexed, experiments exploring its deterministic properties will be established. When these properties are experimentally confirmed or disconfirmed, indexing will place the relation in one of the classes defined above (one-to-one, many-to-one, etc); based on this indexing, new daemons may fire to suggest new experiments or construct new relations or other definitions.

Beyond determinism properties, which apply to any pairing, some properties apply only to relations among a particular type; such relations are pairings whose two sides come from the same source. The properties possibly peculiar to them are algebraic properties: reflexivity, symmetry etc.

6.5 Relational Cliches

Certain pairings qualify as *relations* over particular types. A pairing is a relation if its right and left members may coincide; i.e. if their right and left constraints subsume each other. We can define the meta-type of relations by constructing a mapping function which

gets the right and left constraints of a type:

```
;;; This returns a list of the left and right constraints on a type.
(define (pair-constraints pair-type)
  (list (left-constraints pair-type) (right-constraints pair-type)))
;;; And we declare this as a mapping from pairings to pairs of types:
(declare-mapping! pair-constraints
   pairings (cross-product types types))
```

and use the TYPICAL types [Subsumes Relation] and [Subsumed By Relation] which are satisfied by pairs of types which subsume and/or are subsumed by each other. Given these types and the mapping PAIR-CONSTRAINTS, we can define the type [Relations]. Relations are pairings which relate items from the same space:

```
(define relations
  (<OR> (image-constraint pair-constraints subsumes-relation)
        (image-constraint pair-constraints subsumed-by-relation)))
```

Relations are the connectives of a domain vocabulary; the algebraic properties of a given relation point the way to new representational definitions. Symmetry, transitivity and other regular properties provide the structure around which closures, compositions, and equivalence classes may be provided. In this section, we describe how **Cyrano** uses the confirmation mechanism implemented in TYPICAL to recognize relations as reflexive, anti-reflexive, symmetric, or anti-symmetric.

6.5.1 Recognizing Reflexive Relations

We begin with the definition of [Relations] above and first define the class of reflexive relations. A relation r is reflexive if for all x, $r(x,x)$ is true. The constraint being used here is the identity constraint between the left and right hand sides of a r. A first theory of confirming reflexive relations might be to see if the reflexive relation contains the identity relation $x = x$. If it does — if every equal pair satisfies $x \sim y$ — then the relation is reflexive. Such a definition would look like:

```
(define reflexive-relations
  (empirical-class relations  ; Defined beneath RELATIONS
    (lambda (r) equal-pairs)  ; The sample space generator
    (lambda (r) r)))          ; The evidence space generator
```

The difficulty with this definition is that the reason a given pairing might fail to be in r might be that it doesn't make sense to be related by r. If r implemented the predicate $loves(x,y)$ between *people*, the pair $\langle rock_1, rock_2 \rangle$ would fail the predicate, but that should not affect the reflexivity of *loves*. To remedy this, we define a function RELATION-SPACE which returns the isolated left and right hand constraints of a relation, combined into a single pair constraint:

```
(define (relation-space r)
  (cross-product (left-constraints  r) (right-constraints r)))
```

And then define reflexivity as:

```
(define reflexive-relations
  ;; Defined beneath the meta-type RELATIONS:
  (empirical-class relations
    ;; The new sample space: relevant equal pairs
    (lambda (r) (<AND> equal-pairs (relation-space r)))
    ;; The evidence space is still the relation being tested.
    (lambda (r) r)))
```

Any relation might be reflexive; there are no special heuristics for suspecting that a particular relation might be reflexive or not. Thus, we simply attach to the type Relations the daemon which sets up experiments for reflexivity:

```
(add-daemon! relations
   (lambda (rel) (hypothesis rel reflexive-relations)))
```

A relation may also be *anti-reflexive*; e.g. $r(x, x)$ may never be true. To detect this, we simply make the evidence space be the complement of r, rather than r:

```
(define anti-reflexive-relations
  ;; Defined beneath the meta-type RELATIONS:
  (empirical-class relations
    (lambda (r) (<AND> equal-pairs (relation-space r)))
    (lambda (r) (complement r))))
```

A relation which is reflexive may not be anti-reflexive; this piece of information gives us a heuristic about when to look for a relation being anti-reflexive. If a relation fails to be reflexive, see if it is anti-reflexive. We can encode this by attaching to Not(Reflexive Relations) the experimentation daemon which sets up the experiment for a relation being anti-reflexive:

```
(add-daemon! (complement reflexive-relations)
             (lambda (rel)
                (hypothesis rel anti-reflexive-relations)))
```

Recognizing reflexivity provided an early suprise in the development of **Cyrano**. An early version of **Cyrano** was given a definition of list-equality which used — opaquely — the SCHEME predicate EQUAL?; the confirmation mechanism examined examples and non-examples of this predicate to determine — eventually — that list-equality was reflexive. Later, after considerable development of TYPICAL (and the specification of the INDUCTIVE-DEFINITION combinator (Section 4.5, Page 33)), list-equality was reintroduced as a inductive definition about which TYPICAL — at definition time — made various inferences. When **Cyrano** finally ran again in the extended TYPICAL, the 'confirmation by proof' checkers in HYPOTHESIS fired (as they had never before) on the definition of list equality. Frantically, I searched for the bug before I realized that TYPICAL had actually made the appropriate inferences allowing **Cyrano** to see $\therefore$ from the definition of list equality — that it was neccessarily reflexive. (It contained identity as a base case.)

6.5.2 Recognizing Symmetric Relations

Another property of relations is their symmetry; whether it is always the case that either $r(x,y) \longrightarrow r(y,x)$ or $r(x,y) \longrightarrow \neg r(y,x)$. In the former case, it means that the relation is symmetric and consistently reversible; in the latter case, it means it is an anti-symmetric relation which may establish an ordering of the elements it relates. Both of these relations are symmetries; one is a symmetry of r, the other of $\neg r$.

Symmetry can be considered an invariance of a relation under permutation; it is this interpretation which will guide our definition of the empirical classes for symmetric and anti-symmetric relations. In particular, we introduce a mapping function `TWISTER` (which you may remember from a brief example in the introduction of empirical classes) taking a pair $\langle x,y \rangle$ and permuting it into the mirror image pair $\langle y,x \rangle$. The invariance asserted by the symmetry of a relation R is that pairs satisifying R still satisfy R under permutation. The sample space is then the permuted version of R; the evidence space is R itself. We can thus define the empirical class of symmetric relations:

```
(define symmetric-relations
   ;; Defined beneath the meta-type RELATIONS:
   (empirical-class relations
      ;; The relation type REL, permuted.
      (lambda (rel) (image-constraint twister REL))
      ;; The relation type REL, itself.
      (lambda (rel) rel)))
```

The definition of anti-symmetric relations is analogous to the definition of anti-reflexive relations; we complement the evidence space:

```
(define anti-symmetric-relations
   ;; Defined beneath the meta-type RELATIONS:
   (empirical-class relations
      ;; The relation type REL, permuted.
      (lambda (rel) (image-constraint twister REL))
      ;; The relation type not(REL).
      (lambda (rel) (complement rel))))
```

Since $x = x$ is symmetric, reflexivity is a requirement of symmetry.[11] To avoid performing foredoomed symmetry experiments, we can place the daemon suggesting symmetry on the empirical class of reflexive relations:

```
(add-daemon! reflexive-relations
   (lambda (rel) (hypothesis rel symmetric-relations)))
```

However, the question of where to put the daemon for anti-symmetric experiments reveals a whole in our definition of anti-symmetric relations: reflexive relations *or* anti-reflexive relations cannot be *anti-symmetric*. We must redefine the sample space for anti-

[11] This is not strictly true, but most symmetric and non-reflexive relations are not particularly interesting.

symmetric relations to exclude EQUAL Pairs, implementing the $x \neq y$ constraint in the definition of anti-symmetric relations:[12]

```
(define anti-symmetric-relations
   (empirical-class relations     ; Defined under RELATIONS
      ;; The relation type REL, permuted, with x = x removed.
      (lambda (rel) (<AND> (complement EQUAL-PAIRS)
                           (image-constraint twister REL)))
      ;; The relation type not(REL).
      (lambda (rel) (complement rel))))
```

We can add the daemon for checking anti-symmetry to Relations:

```
(add-daemon! relations
   (lambda (rel) (hypothesis rel anti-symmetric-relations)))
```

Though it might be heuristically advisable to attach it to a more specific class, for instance the union of reflexive and anti-reflexive relations.

Finally, because of our broad definition of relations (any set of pairs extracted from some space), our implementation of confirmation may be 'over eager' in proposing experiments on definitions which have no real inherent semantics (for instance, the type of all pairs of integers, of lists, etc); in the actual implementation of **Cyrano**, only a small class of objects declared 'promising' actually trigger the confirmation process. **Cyrano**'s actual analysis — based on daemons attached to particular subtypes of these 'promising' objects — is more constrained than the promiscuous daemons defined in the examples here.

6.6 A Note on Pragmatics

In the sections above, I have described how TYPICAL is used in **Cyrano** to confirm empirical properties by defining classes of examples and counterexamples. This entire process assumes that there is some source in the world which will provide and index instances of these types so as to drive confirmation or disconfirmation. In some learning systems, this might be a teacher, so we might be able to dispell the requirement of a source of examples by replacing it with 'ask the teacher.'

But a discovery program should be able to do its own experiments. In drawing a parallel to scientific experimentation, it is one thing to say "if X is true when Y is true, we can say Z." On the other hand, the work of science, the everyday labor of the scientist or technician is to construct situations in which X will be true and Y will be observable. A discovery program must also be a problem solver able to generate the potential examples and counterexamples of spaces it has characterized.

The manner in which the **Cyrano** program does this is too involved and too tentative for detailed presentation here. However, the current scheme can be sketched in brief. Every

[12]For purposes of generality, the same modification could be made to the definition of symmetric relations, but we do not do so here. In fact, the space of non-reflexive, symmetric relations doesn't seem to have very many interesting examples.

type has a set of example generators which may be called to provide instances of the type; in turn, these example generators may ask other types for examples, forming a network of example generators.

When a type is asked for an instance, it attempts to call one of its generator methods on the instance; these methods may ask other types for instances which are then combined to construct the requested instance. This generation process may fail; if the chosen method fails to produce a valid instance of a type, one of a small cache of generated instances is selected instead.

In the current implementation, method selection is done at random. It is easy to imagine a more selective criterion which would dynamically order methods based on their effectiveness.

The network of generation methods is determined by the example generators attached to each individual type. This list of generators is established by indexing daemons attached to metatypes in the lattice. When it is noted that examples are needed of a particular type, the type is declared as an *instance source* and indexed; instance sources are an empirical collection (Section 4.4.2 (Page 32)) defined in TYPICAL's lattice. When the type is indexed, indexing daemons run which add new instance generators to the type. For instance, the check for reflexivity consists of generating pairs of items and seeing if the satisfy a relation. We could defined a daemon for adding such a generator in the following manner:

```
(define (add-eq-pair-combiner to-type)
  (let ((combiner-function (lambda (x) (list x x)))
        (combiner-type
          (<AND> (left-constraints type) (right-constraints type))))
    (add-generator! (make-combiner combiner-function combiner-type)
                    type)))
(add-daemon! add-eq-pair-combiner
             (<AND> (power-set eq-pairs) instance-sources))
```

The procedure MAKE-COMBINER constructs an example generator (a procedure of no arguments which returns instances of a type) from a combining procedure and a list of types whose instances it should combines. The daemon defined above adds a generator for EQ-PAIRS to all subtypes of EQ-PAIRS.

In the declaration of generators in this way, *inhibition* (Section 5.1.2 (Page 42)) of daemons becomes important. Generator daemons have varying degrees of power: a weak generation method for some subtype of pairs would simply try random pairs; a more sophisticated approach might use a known implementation of the pairing (for instance, if it represents a LISP procedure) or other constraints. Inhibition is used to have stronger methods inhibit weaker methods; since indexing uses (by default) a 'run them all' strategy, conflicts and priorities must be hand coded as inhibitor daemons in TYPICAL's lattice.

Chapter A-1
TYPICAL Analysis

In this chapter we analyze the algorithms used by the TYPICAL combinators presented in Chapter 4. In particular, we discuss the soundness, completeness, and complexity of algorithms used for type construction and lattice placement. TYPICAL was designed as a module for constructing new type definitions and answering queries about the relations between these constructed definitions. As a module, it must be both reliable and predictable in the fulfillment of its contract; for this reason it is important to have some notion of its correctness (particularly, its soundness and completeness) and its complexity (particularly its decidability and tractablity). This chapter addresses those issues. We begin by providing a semantics for TYPICAL and show where the implemented algorithms fulfill or fail to fulfill these semantics. We then consider the complexity of these algorithms and demonstrate the intractability of completely satisfying the semantic model.

We ignore TYPICAL's synthetic types because their implementation and semantics are generally not dependent on TYPICAL's implementation. In general, a new synthetic type is an opaque predicate defined beneath an existing type by a user or program. The algorithms and mechanisms of TYPICAL only come into play when this type is analytically combined with other types. Thus we restrict our analysis to such analytic combinations.

A-1.1 TYPICAL Semantics

In describing the semantics of TYPICAL we introduce a model containing a universe U of distinguishable points, a set of types T, a set of functions F from U to U, a partial order $\prec$ between types, and a relation ε between elements of the universe and types. The first of these relations represents subsumption and the second represents satisfaction of types by objects in the universe. Satisfaction and subsumption are connected in the obvious way: the semantics of the subsumption relation are those of satisfaction implication. In particular, for any types t_1 and t_2,

$$t_1 \prec t_2 \longleftrightarrow \forall x \in U : \{x \,\varepsilon\, t_1 \longrightarrow x \,\varepsilon\, t_2\}$$

For purposes of describing power sets and meta-types, we further introduce a subset U_T of U denoting types and an accompanying dentotation mapping $D : U_T \Rightarrow T$.

By the nature of implication, we can show several things about the subsumption relation ($\prec$) in this model. Since implication is transitive, so is subsumption:

$$\forall t_1, t_2, t_3 \in T : t_1 \prec t_2 \wedge t_2 \prec t_3 \longleftrightarrow (t_1 \prec t_3)$$

By the reflexivity of implication, subsumption is also reflexive:

$$\forall t \in T : t \prec t$$

Given this model, we can describe the semantics of TYPICAL's analytic combinators. When we prove the soundness of TYPICAL's inferences we will use these definitions. The semantics of TYPICAL's direct type combinators (intersection, union, and complementation) are:

$$\forall x \in U, t_1, t_2 \in T : x \,\varepsilon\, and(t_1, t_2) \longleftrightarrow \forall x \in U, t_1, t_2 \in T : x \,\varepsilon\, t_1 \wedge x \,\varepsilon\, t_2$$

$$\forall x \in U, t_1, t_2 \in T : x \,\varepsilon\, or(t_1, t_2) \longleftrightarrow \forall x \in U, t_1, t_2 \in T : x \,\varepsilon\, t_1 \vee x \,\varepsilon\, t_2$$

$$\forall x \in U, t \in T : x \,\varepsilon\, complement(t) \longleftrightarrow \forall x \in U, t \in T : \neg(x \,\varepsilon\, t)$$

The direct combinators in TYPICAL define a boolean algebra and — as we shall see — it is from this algebra that the implementation's fundamental incompleteness arises.

The indirect type combinators of TYPICAL get their semantics from the functions they are defined in terms of. The semantics of image constraints, for mappings from the space of functions F, are:

$$\forall x \in U, t \in T, f \in F : x \,\varepsilon\, (\boxed{\xrightarrow{f} t}) \longleftrightarrow f(x) \,\varepsilon\, t$$

The semantics for power-sets are similar, except that the denotation function D is used to map type descriptions into subsumption space:

$$\forall t \in T, x \in U : x \,\varepsilon\, PowerSet(t) \longleftrightarrow (x \in U_T \wedge D(x) \prec t)$$

Another possible model for TYPICAL's semantics is an interpretation of types as subsets of a universe U and subsumption as set containment. Early exploration of this model ran into problems due to the representation of power sets and self-containing sets; after it was abandoned (and the semantics above taken up), David McAllester pointed out that this problem could be avoided by introducing a function from types to sets (similar to the

denotation function D above) to serve as a model relation between a set of types and a set of possible models. In practical use, I have found the semantics of satisfaction implication — as opposed to set containment — intuitively preferable in actually analyzing new combinator definitions.

A-1.1.1 Soundness of Direct Type Combinators

We begin by analyzing the direct types of Section 4.1: intersections and unions. As mentioned there, the algorithms used by TYPICAL are not complete; a tradeoff between completeness and tractability was made in desigining TYPICAL since complete subsumption of intersecctions and unions would be NP-hard. Given that we cannot have completeness, we can still examine the soundness of the direct type combinators. The proofs in this section are straightforward and unsuprising; they show that TYPICAL's polynomial time algorithms are sound (but not complete) in generating neccessary implications in the boolean algebra determined by the intersection, union, and complement combinators.

Recalling the algorithms of Section 4.1, inferences for both intersections and unions are made by searching sub-lattices for V-merges or M-merges. The specializations of an intersection and the generalizations of union are found by searching for types which are both below or above (respectively) the intersection or union being created. The generalizations of an intersection and the specializations of a union are found by searching for types which intersect nodes below or union nodes above the type being created. Below we show that these mechanisms — as used by TYPICAL — find types that are appropriate generalizations or specializations of the type being created.

A-1.1.1.1 Specializations of Intersections

The algorithm (described in Section 4.1.2, Page 24) for finding specializations of an intersection type is a search algorithm and we will show that the types found in the search are in fact valid specializations given the semantics of subsumption presented in Section A-1.1.

To find the specializations of an intersection, we search for the V-merges below it in the lattice. This search descends the lattice beneath one of the types being intersected, looking for types which are beneath the other type being intersected. Suppose we are intersecting two nodes t and t' and descend the lattice from t. At some point, we reach the node v. By the semantics of subsumption we know that t subsumes v or (in terms of satisfaction relations):

$$\forall u \in U : u \Sigma v \longrightarrow u \Sigma t$$

At each v encountered, we accept v as a specialization only if it is also under t'; again by subsumption, we know that:

$$\forall u \in U : u \Sigma v \longrightarrow u \Sigma t'$$

```
To find M-merges above and(a,b):
   Make a set of marked nodes M;
   Make a set of m-merges J;
   For every superior s of a or b,
      process the node s;
   To process a node n:
     add n to a set of marked nodes M;
     for each inferior merge i of n:
         if i is an intersection of two nodes in M
               (i.e. they are both marked),
            add i to J and mark the node i;
```

Figure A-1-1. The algorithm for finding M-merges of two nodes marks all the generalizations of two nodes and looks beneath them for nodes which merge marked nodes. (This is a copy of Figure 4-4.)

if the satisfaction of both is implied, the satisfaction of their conjunction is implied. We conjoin the right hand sides of the implications to get:

$$\forall u \in U : u \,\Sigma\, v \longrightarrow (u \,\Sigma\, t \wedge u \,\Sigma\, t')$$

into which (by the if and only if of $and(t, t')$'s semantics) we can substitute:

$$\forall u \in U : u \,\Sigma\, v \longrightarrow u \,\Sigma\, and(t, t')$$

or by the definition of subsumption:

$$v \prec and(t, t').$$

A-1.1.1.2 Generalizations of Intersections

Now we turn to the generalizations for intersections, found by the marking algorithm described in Section 4.1.2 (Page 26). This algorithm is described in Figure A-1-1.

To find the generalizations of an intersection, we begin with the two types being intersected and search for M-merges and direct generalizations above them. The algorithm used is the marking algorithm described in Figure 4-4. To prove the soundness of this algorithm, we use induction on the set of marked types and show that any member of this set is a valid generalization of the new type being defined. We define the set G as those valid generalizations of a type $and(x, y)$; G is defined by by taking the definition of type subsumption in terms of satisfaction and applying it to an intersection type $and(t, t')$ and a potential generalization g:

$$G \equiv \{g : \forall u \in U : u \,\Sigma\, and(t, t') \longrightarrow u \,\Sigma\, g\}$$

We will show that any node marked by our algorithm is in fact a proper member of G.

With the algorithm given in Figure A-1-1, a node is marked as a generalization of $and(t, t')$ if it is t, t', the generalization of a marked node, or an intersection of two marked specializations. We will show that in each of these cases, the nodes marked are in G.

We first show that t and t' are in G. By the definition of intersection,

$$\forall x \in U, t_1, t_2 \in T : x \,\Sigma\, and(t_1, t_2) \longleftrightarrow x \,\Sigma\, t_1 \wedge x \,\Sigma\, t_2$$

we take the left to right implication, substitute in t and t', and break the conjunction to get:

$$\forall u \in U : u \,\Sigma\, and(t, t') \longrightarrow u \,\Sigma\, t$$
$$\forall u \in U : u \,\Sigma\, and(t, t') \longrightarrow u \,\Sigma\, t'$$

placing t and t' in G.

To place the generalizations of marked nodes in G, we show for any $g \in G$ and any type $\tau \in T$, if $g \prec \tau$ then $\tau \in G$. We get this by taking the definition of subsumption:

$$g \prec \tau \longleftrightarrow \forall u \in U : u \,\Sigma\, g \longrightarrow u \,\Sigma\, \tau$$

to get, for any t above g, the implication:

$$\forall u \in U : u \,\Sigma\, g \longrightarrow u \,\Sigma\, \tau$$

which when taken along with the definition that $g \in G$:

$$\forall u \in U : u \,\Sigma\, and(t, t') \longrightarrow u \,\Sigma\, g$$

can be chained to show (by the definition of G) that $\tau \in G$:

$$\forall u \in U : u \,\Sigma\, and(t, t') \longrightarrow u \,\Sigma\, \tau$$

This justifies marking all of the generalizations τ of t and t'. Finally, we consider the marking of joins beneath these generalizations.

If a join $and(\tau, \tau')$ is marked it means that both τ and τ' are marked. Therefore, we know that they are both in G:

$$\forall u \in U : u \,\Sigma\, and(t, t') \longrightarrow u \,\Sigma\, \tau$$
$$\forall u \in U : u \,\Sigma\, and(t, t') \longrightarrow u \,\Sigma\, \tau'$$

Further, if both are satisfied, the conjunctive statement is true:

$$\forall u \in U : u \,\Sigma\, and(t, t') \longrightarrow (u \,\Sigma\, \tau \wedge u \,\Sigma\, \tau')$$

where the consequent is the same as satisfaction of the type intersection $and(\tau, \tau')$:

$$\forall u \in U : u \,\Sigma\, and(t, t') \longrightarrow u \,\Sigma\, and(\tau, \tau')$$

placing $and(\tau, \tau')$ firmly in G as a valid generalization of $and(x, y)$. Thus, all types marked by the algorithm are in fact generalizations of the intersection $and(t, t')$.

Finding the generalizations and specializations of a union are the exact mirror of the process for an intersection. We look down for M-merges and up for V-merges.

A-1.1.1.3 Generalizations of Unions

To find the generalizations of a union, we search for the V-merges above it in the lattice. This process is simply the dual of the search for specializations of an intersection, but its

soundness proof is included for completeness. This search ascends the lattice above one of the types being intersected, looking for types which are above the other type being intersected. Suppose we are generating the union of two nodes t and t' and ascend the lattice from t. At some point, we reach the node v. By the semantics of subsumption we know that t is subsumed by v or (in terms of satisfaction relations):

$$\forall u \in U : u \,\Sigma\, t \longrightarrow u \,\Sigma\, v$$

At each v, we accept v as a generalization only if it is also above t'; again by subsumption, we know that the following satisfaction relation holds:

$$\forall u \in U : u \,\Sigma\, t' \longrightarrow u \,\Sigma\, v$$

if the satisfaction of either x or y implies satisfaction of v, the disjunction satisfies v. We disjoin the left hand sides of the implications:

$$\forall u \in U : (u \,\Sigma\, t \vee u \,\Sigma\, t') \longrightarrow u \,\Sigma\, v$$

and substitute (by the definition of type union):

$$\forall u \in U : u \,\Sigma\, or(t, t') \longrightarrow u \,\Sigma\, v$$

or by the definition of subsumption:

$$or(t, t') \prec v.$$

A-1.1.1.4 Specializations of a Union

To find the specializations of a union, we begin with the two types being unioned and search for M-merges and direct specializations below them. This process is simply the dual of the search for generalizations of an intersection, but its soundness proof is included for completeness. The algorithm used is a straightforward adaption of the marking algorithm given in Figure A-1-1, but with 'above' replaced by 'below,' 'intersection' replaced by 'union' and so forth. To prove the soundness of this algorithm, we again use structural induction on the set of marked types and show that any member of this set is a valid specialization of the new type being defined. We define the set S as those valid specializations of a union $or(x, y)$; S is defined by taking the definition of type subsumption in terms of satisfaction and applying it to an arbitrary union type $or(t, t')$ and some potential specialization s:

$$S \equiv \{s : \forall u \in U : u \,\Sigma\, s \longrightarrow u \,\Sigma\, or(x, y)\}$$

We will show that any node marked by our algorithm is in fact a proper member of S.

With our modification of the algorithm of Figure A-1-1, a node is marked as a specialization of $or(t, t')$ if it is t, t', the specialization of a marked node or a union of two marked specializations. We will show that in each of these cases, the nodes marked are in S.

We first show that t and t' are in S. By the definition of union we know that, for $or(t, t')$:

$$\forall x \in U : x \,\Sigma\, or(t, t') \longleftrightarrow (x \,\Sigma\, t \vee x \,\Sigma\, t')$$

which we can break into independent implications

$$\forall u \in U : u \,\Sigma\, t \longrightarrow u \,\Sigma\, or(t, t')$$
$$\forall u \in U : u \,\Sigma\, t' \longrightarrow u \,\Sigma\, or(t, t')$$

placing t and t' in S.

To show that specializations of marked nodes have a place in S, we show for any $s \in S$ and any type $\tau \in T$, if $\tau \prec s$ then $\tau \in S$. We get this by taking the definition of subsumption:

$$\tau \prec s \longleftrightarrow \forall u \in U : u \,\Sigma\, \tau \longrightarrow u \,\Sigma\, s$$

and use the fact that s is marked ($s \in S$):

$$\forall u \in U : u \,\Sigma\, s \longrightarrow u \,\Sigma\, or(t, t')$$

and chain the implications to show that τ satisfies the definition of membership in S:

$$\forall u \in U : u \,\Sigma\, \tau \longrightarrow u \,\Sigma\, or(t, t')$$

This justifies marking all of the specializations τ of t and t'. Finally, we consider the marking of or-merges above these specializations.

If a join $or(\tau, \tau')$ is marked it means that both τ and τ' are marked. Therefore, we know that they are both in S:

$$\forall u \in U : u \,\Sigma\, \tau \longrightarrow u \,\Sigma\, or(t, t')$$
$$\forall u \in U : u \,\Sigma\, \tau' \longrightarrow u \,\Sigma\, or(t, t')$$

which is the same as the single implication from their disjunction:

$$\forall u \in U : (u \,\Sigma\, \tau \vee u \,\Sigma\, \tau') \longrightarrow u \,\Sigma\, or(t, t')$$

where the antecedent is the same as satisfaction of the union type $or(\tau, \tau')$:

$$\forall u \in U : u \,\Sigma\, or(\tau, \tau') \longrightarrow u \,\Sigma\, or(t, t')$$

placing $or(\tau, \tau')$ firmly in S as a valid specialization of $or(t, t')$. Thus all types marked by TYPICAL's algorithm are in fact valid specializations of $or(t, t')$.

A-1.1.1.5 Generalizations and Specializations of Complements

The types above and below a complement are computed by searching the lattice above and below the type being complemented for types which have defined complements. These complements are respectively below and above the complement being defined. To show soundness we consider the subsumption relation between two types t and t' with complements $complement(t)$ and $complement(t')$. If we know that t is below t' in the lattice ($t \prec t'$), we know by the definition of subsumption that:

$$\forall u \in U : u \,\Sigma\, t \longrightarrow u \,\Sigma\, t'$$

by inverting the implication:

$$\forall u \in U : \neg(u \,\Sigma\, t') \longrightarrow \neg(u \,\Sigma\, t)$$

we see that if t' is ever unsatisfied, t must be unsatisfied. We also know from the definition of the complementation combinator that failure of satisfaction implies satisfaction of the complement and vice versa; thus satisfaction of $\mathit{complement}(t')$ will imply satisfaction of $\mathit{complement}(t)$, showing that

$$\mathit{complement}(t') \prec \mathit{complement}(t').$$

In the same way, we can show the completeness of such inferences, for if

$$\mathit{complement}(t') \prec \mathit{complement}(t'),$$

we know by the definition of complementation, that

$$\forall u \in U : \neg(u \sqsupset t') \longrightarrow \neg(u \sqsupset t)$$

or that

$$\forall u \in U : u \sqsupset t \longrightarrow u \sqsupset t'$$

which is the same as saying that $t \prec t'$. This completeness is only a partial result; the inferences it makes are complete only if the rest of the lattice is complete, and as we will see below, this is not the case.

A-1.1.1.6 Incompleteness Results

Above we showed the *soundess* of the algorithms used by TYPICAL to make inferences about intersections, unions, and complements. In the case of intersections and unions we did not show completeness, for the algorithms used by TYPICAL are not complete; there exist valid satisfaction inferences which are not identified as subsumptions by the algorithm. The problem of subsumption with intersections and unions is intractable and it would thus be unlikely that the algorithms above are complete. In fact, there exist counterexamples of valid inferences which the algorithms do not make; the nature of such counterexamples — the holes in the algorithm — are described in this section.

One incompleteness lies in the implementation of complements. It is clear that the union of a type and its complement should be equivalent to the top of the lattice

$$t \vee \mathit{complement}(t) \equiv \top$$

but TYPICAL does not make this inference nor any other inferences that depend on the knowledge that two types are complements or even disjoint. From the point of view of the subsumption inference algorithms, the complement of a type is simply a primitive type which is related to other complements in a particular way.

Another incompleteness lies in the interaction of intersection and union types in the lattice. The expressions

$$(A \vee B) \wedge (B \vee C)$$
$$(B \vee (A \wedge C))$$

are logically equivalent. However, the algorithms used by TYPICAL only discover subsumption of the second beneath the first and not vice-versa (which is also a valid inference).

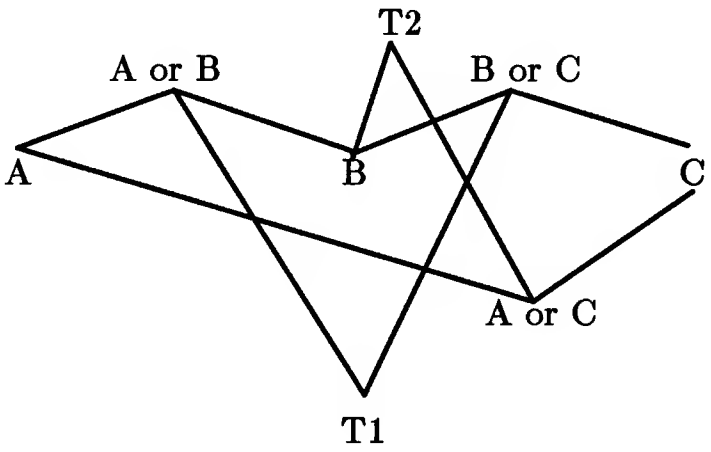

Figure A-1-2. TYPICAL's inference algorithms are not complete; the node T1 is logically identical to T2 and should thus be both above and below T2 in the lattice.

Figure A-1-2 shows a lattice fragment corresponding to these expressions. If we construct the type corresponding to the first expression, it is the intersection of two unions; the first union has A and B as specializations and the second has B and C as generalizations. The resulting intersection is placed (by V-merge search) above B. When we construct the second expression, it is beneath the first by subsumption through B, but there is not any connection placing B below it. While sound, the inference algorithms used by TYPICAL are not complete. This is where the boolean algebra implemented by TYPICAL's direct combinators (and mentioned at the beginning of the chapter) falls flat; however, as we will show towards the end of the chapter, a complete implementation would be computationally intractable.

A-1.1.2 Soundness of Indirect Type Combinators

The direct type combinators generate types for which satisfaction is determined by the simple application of the types they combine; thus, they are related — by satisfaction implication — directly to those types. Types generated by the indirect combinators, on the other hand, translate the object being tested into another 'space' before applying the types they combine; thus, they are related by subsumption to types defined in this other space. For instance, image constraints find subsumption relations in a type space dividing the range of the mapping used by the image constraint; power sets find subsumption relations among the meta-types which divide the space of known types.

In TYPICAL, Both power sets and image constraints operate by annotating their inputs with the outputs they produce. For instance, if an indirect combinator l is applied to a type x to produce a type $l(x)$, the type x is annotated with $l(x)$. In generating an $l(x)$,

the lattice above and below x is searched for other generated types $l(y)$ by checking each y above or below x. These will be placed above and below the newly created type $l(x)$.

A-1.1.2.1 Analysis of Power Sets

For example, in generating a power set of a type t, the superiors and inferiors of t are searched for already defined powersets. We will prove the soundness of this algorithm by show that the power set mapping preserves the subsumption relation between power sets and the types they are defined for. We begin by specializing the relation between subsumption and satisfaction to powersets in particular:

$$PowerSet(\tau) \prec PowerSet(\tau')$$

iff

$$\forall u \in U : u \,\Sigma\, PowerSet(\tau) \longrightarrow u \,\Sigma\, PowerSet(\tau')$$

and note that satisfaction of a powerset is equivalent to subsumption after mapping through the denotation function D:

$$\forall t \in T, u \in U' : u \,\Sigma\, PowerSet(t) \longleftrightarrow D(u) \prec t$$

given that u is in U' (the domain of D); we can practically ensure this by placing created power sets under the meta type $\boxed{\text{Types}}$ corresponding to U'. Transforming the subsumption statement, we get:

$$PowerSet(\tau) \prec PowerSet(\tau')$$

iff

$$\forall u \in U' : D(u) \prec \tau \longrightarrow D(u) \prec \tau'$$

since we know that for any $t \in T$, there exists an $x \in U'$ such that $D(x) = t$, we can change the right hand side into a claim about T:

$$PowerSet(\tau) \prec PowerSet(\tau')$$

iff

$$\forall t \in T : t \prec \tau \longrightarrow t \prec \tau'$$

or, by the definition of subsumption ($\succ$):

$$PowerSet(\tau) \prec PowerSet(\tau')$$

iff

$$\tau \prec \tau'$$

A-1.1.2.2 Analysis of Image Constraints

In the case of image constraints, the approach is more or less the same; the distinction is that image constraints entail a class of annotations, one for each mapping. For instance, if we consider the type of lists whose first elements are integers ($\boxed{\xrightarrow{CAR} Integers}$),

its generalizations will be those **CAR** constraints whose image lies above $\boxed{\text{Integers}}$ and its specializations will be those **CAR** constraints whose image lies below $\boxed{\text{Integers}}$. We begin with the equivalence of subsumption and satisfaction implication:

$$\boxed{\xrightarrow{f} t_1} \prec \boxed{\xrightarrow{f} t_2} \longleftrightarrow (\forall x \in U : f(x) \,\Sigma\, t_1 \longrightarrow f(x) \,\Sigma\, t_2)$$

and since we know that every f maps into U, we can use the implication:

$$(\forall x \in U : x \,\Sigma\, t_1 \longrightarrow x \,\Sigma\, t_2) \longrightarrow (\forall x \in U : f(x) \,\Sigma\, t_1 \longrightarrow f(x) \,\Sigma\, t_2)$$

where the right hand side above is equivalent to subsumption between t_1 and t_2; we chain this with the right-to-left implication above to get:

$$t_1 \prec t_2 \longrightarrow \boxed{\xrightarrow{f} t_1} \prec \boxed{\xrightarrow{f} t_2}$$

to get the soundness of the image constraint algorithm.

However, the algorithm for image constraints is not complete because the mapping function f may introduce structure among image constraints which does not occur among the constraining types. For an extreme case, consider the case where f is a constant function $f(x) = C$. In this case, any image constraints using f are equivalent and thus subsume each other; for any t_1 and t_2, $\boxed{\xrightarrow{f} t_1} \prec \boxed{\xrightarrow{f} t_2}$ and $\boxed{\xrightarrow{f} t_2} \prec \boxed{\xrightarrow{f} t_1}$. Since our algorithm does not take any information about f into account, inferences based on such 'introduced structure' (like $f(x) = C$) cannot be made. However, we can show that any inferences beyond those made by TYPICAL requires knowledge about the properties of f, something which TYPICAL, at least, makes no attempt to represent.

Suppose that we had an algorithm which could make a correct subsumption inference about a type $\boxed{\xrightarrow{f} P}$ that placed it under a type $\boxed{\xrightarrow{f} Q}$, where $P \not\prec Q$; such an algorithm would be more powerful than the algorithm used by TYPICAL. This subsumption inference, given the definition of image constraints, means that for any u in U:

$$f(u) \,\Sigma\, P \longrightarrow f(u) \,\Sigma\, Q$$

Since $P \not\prec Q$, there is some u' such that $u' \,\Sigma\, P$ and $u' \not\Sigma\, Q$. If f is defined so that for some (or many or all) u, $f(u) = u'$, the subsumption implication

$$f(u) \,\Sigma\, P \longrightarrow f(u) \,\Sigma\, Q$$

will fail for some (or many or all) u implyuing the negation of our assumption. Thus, the algorithm — to be correct — must have the information that f is not defined in such a manner. Intuitively, any additional structure among image constraints — beyond that inferred by TYPICAL from the lattice around the constraining types — must be based on structure *introduced* by the mapping f.

TYPICAL makes two special case inferences based on particular properties of the mapping function f; these use an explicitly declared domain and range of f to catch two special cases. The first case is where the constraining type Q of $\boxed{\xrightarrow{f} Q}$ lies outside the range of f, and the type $\boxed{\xrightarrow{f} Q}$ is empty; the second case is where the constraining type P is exactly

the range of f and hence the type $\boxed{\xrightarrow{f} P}$ is simply the domain of f. In fact, all image constraints around f are under the domain of f. TYPICAL 'implements' the first special inference by disallowing the definition of such types; the second inference is handled by placing any image constraint based on a mapping f under the domain of f.

A-1.1.2.3 Relative Completeness

In the above proofs, we showed that the algorithms for finding subsumption between intersections, unions, and complements were sound but not complete; discovery by the algorithm was sufficient but not neccessary for an actual subsumption relation to hold. In the case of power sets and complements, we found a complete equivalence between subsumption relations in one part of the lattice and another. And in the case of image constraints, the algorithm was found to be as complete as possible in the absence of detailed knowledge about the image mapping. Given that the rest of the lattice is complete and that image mappings add no special structure, these algorithms — for complements and indirect types — are complete. Even though the complete implementation of TYPICAL is not complete, it is still possible for us to speak of the 'relative completeness' of these individual sub-modules.

The fundamental incompleteness of TYPICAL (outside of the 'empirical' incompleteness introduced by arbitrary mapping functions) comes from the incompleteness of subsumption inferences between intersections, unions, and complements. While there exist provably complete algorithms for such inferences (for instance, translation into boolean satisfaction), TYPICAL's design has made a compromise between completeness and complexity which bars such approaches. The resulting complexity — and the intractability of a complete solution — are described in the remainder of this chapter.

A-1.2 TYPICAL Complexity

The contract of TYPICAL characterizes two isolable functions: the definition of new types and queries about the relations between types.[13] Often, these may be interleaved, but as transactions with TYPICAL *qua* module, they may be isolated. A decision was made early in TYPICAL's design to make queries very fast at the expense of time and space in the construction of new types. With the lattice cache described in Section 3.1 (Page 17), queries to the lattice about existing types are computable in constant time.[14] The task

[13]This is not quite true. Another important function of TYPICAL is the satisfaction query: asking whether a given object satisfies a given type. However, the complexity of this query is not isolable to a particular type defintion, but rather depends on the complexity of TYPICAL's primitive definitions and their procedural combination.

[14]Assuming the standard caveats about a finite memory machine with constant time address decoding. In all of the following discussion, we will assume that referencing bit vectors and

of analysis then turns to the complexity of the combinators used to define new types, in particular their computation of generalization and specialization relations in the lattice.

In analyzing the complexity of TYPICAL's combinators, it is useful to define two particular sorts of sublattices of the lattice: the sublattice $G^*(x)$ above a type x and the sublattice $S^*(x)$ below a type x. We will also occasionally refer to the intersection and union of these sublattices; when we do so these will denote the lattice generated by the union of both edges and nodes in the sublattices. Most of the algorithms presented in the previous sections operate on these sublattices, searching in them for types which — directly or under some trivial transformation — are generalizations or specializations of a type being defined.

Each of TYPICAL's algorithms analyzed here enumerate such sublattices; to describe their time complexity we will use the notation $E(x)$ to describe the time neccessary to enumerate the sublattice x. In Section A-1.2.3, we will describe the worst and expected case properties of $E(G^*(x))$ and $E(S^*(x))$.

An important point is that in the following sections, we analyze the time taken to find generalizations and specializations in the lattice; this is quite indepenent of the time taken to install these links in the lattice, or — particularly — to update the lattice subsumption cache. With only the standard address space assumptions, the time is $O(E(S^*(t)n))$ where a subsumption link is being established between t and t' and the subsumption cache for t' must be logically OR'd (taking $O(n)$ time) with the subsumption caches for each specialization of t. However, the constraint that type construction only create new relations with the type being constructed allows us to dispense with the $O(n)$ for each OR; instead we need only set one bit corresponding to the new type. This gives us an actual update time of $O(E(S^*(t)))$ (a bound which will become familiar).

Practically, however, most new types are created at the lower fringes of the lattice where $S^*(t)$ will be relatively small; this was the reason that the subsumption cache was chosen to represent generalization rather than specialization, which would require enumerating $G^*(t)$. In general, TYPICAL's algorithms end up enumerating some G^* lattice, so that the cost of finding subsumptions overshadows the cost of installing them.

A-1.2.1 Complexity of Direct Type Combinators

The inferences of the intersection and union combinators work by looking for V-Merges and M-Merges above or below the types being defined. Since there is no interaction between these two processes, the time taken by the combinator is the sum of the time taken to find M-Merges and the time taken to find V-Merges. In the case of intersections, M-merges are looked for in the direction of generalization, while V-merges are looked for in the direction

---

table lookup on types is computable in constant time. This also assumes that we have an upper bound on the size of the table or bit vector; since each describes properties per-node and there are a fixed number of nodes when a type is created, this assumption actually holds.

of specialization. For unions, the opposite holds (since the processes are duals of each other), but the complexity remains the same. In this section we will assume that we are analyzing a type intersection, with the understanding that the same analysis hold for type unions.

The specializations for an intersection of two types are found by looking at the specializations of one for types which are also beneath the other. These types are called V-merges because two connected paths descend from the types being intersected to form a 'V' below the intersection being created. Since determining subsumption is a constant time operation, the time taken to find V-merges is proportional to the time taken to enumerate the sublattice. If the root of the sublattice being searched is x, this is $E(S^*(x))$. The actual time taken may be less than this quantity, since once a V-merge is found on one path through the lattice, the nodes below it will all be beneath (by transitivity) the new intersection and need not be searched. However, with no heuristics for selecting which node of an intersection $intersection(x, y)$ to use as a root for the search, the actual upper bound for finding V-merges will be $max(E(S^*(x)), E(S^*(y)))$.

The algorithm of finding the M-merges of an intersection — its accidental generalizations — is more complicated than that for V-merges. The search for M-merges — described in Section 4.1.2 (Page 26) — proceeds by climbing the lattice and then 'looking down' for M-merges. This can be seen as looking for types for which the type being defined would be a V-merge. The algorithm described in Section 4.1.2 and displayed in Figures 4-4 and A-1-1 is a marking algorithm which — given a pre-determined bound on the number of nodes — takes constant time at each node. As with the V-merge algorithm, the time taken to search for M-merges is proportional to the number of edges traversed in the marking algorithm. But unlike the search for V-merges, this size is not proportional to the size of any particular sublattice, since each node in the subgraph $G^*(x) \cup G^*(y)$ might be expanded into an entire sublattice. The time taken for an M-merge search is thus proportional to the time required to enumerate:

$$\bigcup_{g \in G^*(x) \cup G^*(y)} S^*(g)$$

The size of this set has a lower bound of $G^*(x) \cup G^*(y)$ and, as we will describe later, an upper bound of the number of edges in the lattice.

The algorithm for determining the subsumption relations of a complement searches the generalizations and specializations of the type being complemented for other types which already have complements defined. Since determining whether a type has a complement defined for it is a constant time operation, the time for determining these subsumption relations is simply the time required for enumerating the specializations and generalizations of the type being complemented. This is simply $O(E(G^*(t)) + E(S^*(t)))$.

A-1.2.2 Complexity of Indirect Type Combinators

As for complements, the complexity of indirect combinators depends on the size of sub-

lattices above and below the type which the indirect type is being defined from. The algorithms for indirect types comb this lattice for types annotated with pointers to related indirect types.

If we assume that we can fetch annotations in constant time, the time required for finding specializations of a power set of t is proportional to $E(S^*(t))$ and the time for finding generalizations of a power set is $E(G^*(t))$. These are both upper bounds, as with search for V-merges of direct types, once we find an appropriate constraint on one path through the sublattice, the types beyond it will be included by transitivity, so the search along that path may be terminated.

Finding the generalizations and specializations of image constraints is precisely the same, assuming constant time to fetch the annotation corresponding to a particular mapping and type. The time required for finding specializations is $E(S^*(t))$ and for finding generalizations is $E(G^*(t))$. Just as for power set, these are upper bounds; discovery of matching constraints along a path prunes the rest of the path from the search space.

Given this analysis of indirect types, we can consider what bounds are actually placed on compuations given worst case and average case search times for S^* and G^*.

A-1.2.3 Properties of $E(G^*(x))$ and $E(S^*(x))$.

Each of the above complexity bounds was expressed in terms of the time required to search sublattices G^* and S^*. In this section we examine the nature of these terms. If we could be guaranteed that the fanout of each type in a sublattice l was less than some constant, we could guarantee an upper bound on $E(l)$ proportional to the number of nodes in l. Unfortunately, we have no such guarantee; thus, we may only guarantee that the time $E(l)$ is proportional to the number of edges in l, which has an upper bound of the square of the number of nodes. However, for any given $E(G^*(x))$ or $E(S^*(x))$, this bound will only be reached if the sublattice is the entire lattice and the lattice itself is completely connected (which never happens).

A more interesting question is what average case properties do $E(G^*(x))$ and $E(S^*(x))$ have? Note that now we know that E can just as easily stand for the number of edges in the lattice as the enumeration time. Looking at TYPICAL running **Cyrano** (in April 1987 with 490 types and 3,989 edges in the lattice), we note that the average sublattice consists of 9.7 nodes and 44.4 edges. This is considerably better than the $490^2 = 240,100$ given as a worst case above. Of course, the important element here is the linear behaviour of the combinators on the sublattice. In the next section we describe how the tractability of these algorithms was maintained at the cost of completeness in their performance.

A-1.3 Tractability Tradeoffs

In designing TYPICAL, tradeoffs were made between completenes and tractability in making subsumption inferences. Nearly all subsumption problems are intractable in their

complete solution and the development of TYPICAL has been dotted with 'completeness compromises' as more and more of its inferences were shown to be intractable in their most general case. This section shows how most of the inferences involved in subsumption are NP-hard (thus probably intractable), and describes some of the 'holding' positions along the way.

TYPICAL's algorithms enumerate the generalizations and specializations of a newly created type; in the analysis below, we examine a slightly different question: given two types s and g, does s subsume g? If there were a complete algorithm for enumerating generalizations and specializations, it would be able to answer the subsumption question by just checking the list of generated generalizations or specializations; thus the enumeration problem is at least as hard as the subsumption question. If we had an algorithm for the subsumption question, however, we could apply it to all combinations of the newly created type with existing types; thus, the enumeration problem is at most $O(n)$ harder than the subsumption question. Below, we show that the subsumption question is NP-hard by showing that it is co-NP-complete. For this proof we use the known co-NP-complete problem TAUT (determining if a given boolean expression is a tautology) and show that subsumption is reducible to TAUT in polynomial time and that (vice versa) the TAUT question is reducible to the type subsumption question.

A-1.3.1 Intractability with AND, OR, and NOT combinators

This section shows the intractability of the subsumption question given the AND, OR, and NOT combinators. Section A-1.3.3 shows a more general result (due to David McAllester) that demonstrates intractability with simply AND and OR combinators. In both cases, we use reduction to and from the TAUT problem, beginning in the easy direction: showing that the subsumption question is no harder than TAUT and thus is at least in co-NP.

To show that subsumption is in the class co-NP, we will show how to translate a subsumption problem between types s and g into determining if a boolean expression is a tautology, which is known to be in co-NP. The conversion proceeds by translating the types s and g into two boolean expressions S and G. In S and G, each type union is represented as a disjunction, each type intersection as a conjunction, and each complementation as a negation. The primitive terminal types in the definitions of s and g become variables in S and G. The subsumption relations between primitive types are represented by a conjunction L of implications between the corresponding variables (e.g. if a primitive terminal x is beneath a primitive terminal y, the implication $(X \longrightarrow Y)$ is one term in L) This translation can be done in time proportional to the size of the lattice; then, to see if s is beneath g in the lattice we check that $((L \wedge S) \longrightarrow G)$ is a tautology. If so, s must be below g.

This shows that subsumption is in co-NP: solvable in polynomial time given a polynomial time solution to TAUT; to prove the opposite direction we show that solving subsumption in polynomial time will allow us to determine if a boolean expression is a tautology. To

find if an expression E is a tautology, we create a primitive type for each variable in E and use the complement, intersection, and union combinators to define a type corresponding to E. We then see if this type is identical to (both 'above' and 'below') the top of the lattice. Given that type construction is polynomial, the number of types constructed for an expression E must be a polynomial function of the size of finite E. Thus the translation can be done in polynomial time and if we can determine subsumption in polynomial time as well, we can resolve if S is a tautology in polynomial time by composing the processes. Solving subsumption with AND, OR, and COMPLEMENT combinators is co-NP-complete and thus NP-hard.

A-1.3.2 Intractability with AND, OR, and disjointness

Originally, the author believed that weakening representing complementation to representing disjointness would make subsumption tractable. The argument in favor of this began with the assumption that copmlete subsumption with just AND and OR was computable in polynomial time; it then assumed that no new inferences were possible from knowledge of disjointness and thus representing disjointness could not complicate subsumption. Unfortunately, David McAllester managed to prove both of these assumptions false; he found and embarassingly simple example of subsumption inferences from disjointness and also proved that the inference of subsumption relations between type-unions and type-intersections alone is co-NP-complete. A version of these proofs is given below.

The notion that disjointness would not play a role in subsumption inference can be seen from a boolean logic interpretation of type subsumption. In particular, from the knowledge that types A and C are disjoint, we can show that

$$((A \vee B) \wedge (B \vee C)) \longrightarrow B$$
$$and(or(a,b),or(b,c)) \prec b$$

where the logical implication above is equivalent to the new subsumption relation below. While this is not enough to prove intractability, it does prove the incompleteness of any subsumption algorithm which ignores disjointness information.

A-1.3.3 Intractability with AND and OR

The intractability of subsumption given AND and OR is slightly more complicated. We first recognize that the problem is a subset of type subsumption with complementation and is trivially in co-NP. (Since we just demonstrated that subsumption *with* complements is no harder than TAUT.) To show that it is co-NP-complete and thus NP-hard, we will use a modified method of translating expressions into types and show that solving this problem would allow us to determine (as above) if a boolean expression is a tautology.

The key idea in the proof is to replace each variable and its complement by two separate variables and then represent the identity constraints on the variables in a separate expression.

We begin with a function χ with transforms an expression into a type in the following manner:

- Each uncomplemented variable V_i is converted to a primitive type v_i.
- Each complemented variable $\neg V$ is converted to a primitive type v_i'.
- Each disjunction $(A \vee B)$ is converted into a union type $or(\chi(A), \chi(B))$.
- Each conjunction $(A \wedge B)$ is converted into an intersection type $and(\chi(A), \chi(B))$.

Suppose we want to determine if a boolean expression T is a tautology; we first generate $\neg T$ in polynomial time and then — given polynomial time type construction — construct a type $\chi(\neg T)$.

This $\chi(\neg T)$ is a type defined only in terms of intersections and unions of primitive types. We can see that satisfaction of the primitive types (every v_i and v_i') referred to by $\chi(\neg T)$ determines a truth model for the variables in T; this truth model can be viewed as a function specifying the assignment of a variable V based on satisfaction of the simple types v_i and v_i' (which represent V_i and $\neg V_i$ respectively):

$$f(V) \text{ is true} \longleftrightarrow \forall u \in U : u \, \Sigma \, v$$
$$f(V) \text{ is false} \longleftrightarrow \forall u \in U : u \, \Sigma \, v'$$

The expression T will be a tautology if and only if $\neg T$ is true only for invalid models. In order for the model to be a valid model, f must be a true function; this is only possible if it is never the case that v and v' are simultaneously satisfied. We can define a new type q which is satisfied only when the model determined by every v and v' is invalid. This type could be described by:

$$q = \bigvee_{\text{all v}} (v \wedge v')$$

Given polynomial time type construction, we can also construct q in time polynomial in the size of T (since if the size is n, the number of variables must be less than n). Now, if T is a tautology, $\neg T$ is true only if the truth function is invalid. $\neg T$ being true corresponds to $\chi(\neg T)$ being satisfied, so T being a tautology implies that:

$$\forall u \in U : u \, \Sigma \, \chi(\neg T) \longrightarrow u \, \Sigma \, q$$

which is the same as saying $\chi(\neg T) \prec q$. If there existed a polynomial time subsumption algorithm for intersections and unions, we could generate t and q in polynomial time and then determine if $t \prec q$ in polynomial time, thereby finding if an expression T is a tautology in polynomial time. *Subsumption, even with only intersections and unions, is co-NP-complete and thus NP-hard.*

One common false proof of the tractability of AND/OR subsumption is based on the tractability of subsumption between conjunctive normal form expressions without comple-

mentation. If we have two expressions in conjunctive normal form (CNF) without complementation, we can compute subsumption in polynomial time. Given two expressions A and B of the form:

$$(\wedge\ (\vee A_1 A_2 \ldots)\ (\vee A_1 A_3 \ldots)\ \ldots)$$

subsumption can be checked by seeing if the variables of each internal $\wedge$ in A are a subset of the variables in *one* internal $\wedge$ in B. If this is true, subsumption follows. The hole in the proof is that the polynomial time conversion to CNF introduces complements into the CNF version of an expression that started out without complements. David McAllester pointed out that the conversion to CNF without complements is possible, but would take exponential time.

A-1.4 Conclusions

In the above demonstrations we have shown that a complete algorithm for subsumption is NP-hard; in the interests of the tractability desirable in a module, the contract of TYPICAL does not guarantee completeness but only soundness. This is a reasonable tradeoff from the standpoint of TYPICAL's intended application, the inductive discovery program **Cyrano**. **Cyrano** uses TYPICAL as an inference engine for providing *obvious* relations between types; **Cyrano** itself is searching for *accidental* relations between types. The incompleteness of TYPICAL's algorithms means that this pruning will be incomplete; the correctness of the final distinction between accidental and neccessary implications can be maintained by applying a complete exponential time lattice analysis on empiricially discovered subsumption relations. Thus, for **Cyrano**'s purposes, the incompleteness of TYPICAL's inferences is not critical. Other applications of TYPICAL (for instance, to program type analysis) may suffer, but that must be determined for each individual case.

This empty page was substituted for a blank page in the original document.

Chapter A-2
A TYPICAL Manual

This appendix provides a sketchy manual to TYPICAL; it introduces the basic procedures and data structures, as well as the types initially defined by TYPICAL. In addition, it documents a handful of utility functions which TYPICAL uses. Finally, it briefly documents the TYPICAL indexer, as described in Chapter 5, specifying the top level daemons for specifying daemons and inhibitions.

The files defining TYPICAL are described in Appendix A-3; each of the procedures below lists its file of definition for programmers or users who wish to peruse the source code.

A-2.1 Type Descriptions

Type descriptions are implemented as SCHEME vectors. As described in Section 3.1, each type description possesses a unique integer identifier. Each type description prints out as:

`#[`*id*`:`*name*`]`

where *id* is the unique integer identifier for the type. The *name* appearing in a printed representation may come from a variety of sources. By default, the style of the name is based on the primitive combinator which constructed the type.

For instance, types constructed by the `TYPE-INTERSECTION` of two types have a *name* consisting of the types intersected separated by the string `<and>` E.G. the type intersection of `#[12:Men]` and `#[13:Unmarrieds]` would print as:

```
#[14:#[12:Men]<and>#[13:Unmarrieds]]
```

The printed form of a particular type can be specified by the `NAME-TYPE!` or `TYPE` procedures. Evaluating

(`name-type!` *type name*)

Gives `type` the name `name`, where `name` is either a string, a list which can be passed to `PRINTOUT` (Section A-2.9; Page 92), or a procedure of no arguments which prints out a description of the type. The `TYPE` procedure gives a name to a newly created type; it has the form:

(`type` *name combinator ... combinator-arguments*)

which creates a new type by calling *combinator* on *combinator-arguments* to produce a type which is then named by *name*, which is of the same format accepted by `NAME-TYPE`. It finally returns the type created. The `TYPE` procedure is typically used when definiing types at top level, e.g.

```
(define lists (type "Lists" simple-type list? lisp-objects))
```

There are several functions for checking, finding, and accessing type descriptions; the ones described below are the simplest and most useful public ones.

(**TYPE-DESCRIPTION?** *object*) ⟶ *yes-or-no*
Returns `#T` (true) if *object* is a type description.

(**->TD** *object*) ⟶ *type-description*
Attempts to coerces an object into a type description. If *object* is an integer, the type description possessing that integer index is returned; if *object* is a procedure describing a predicate, the type description corresponding to the predicate is returned.

(**TD-PREDICATE** *type-description*) ⟶ *predicate*
Returns the determining predicate of *type-description*.

(**TD-ID** *type-description*) ⟶ *integer-id*
Returns the unique integer identifier for a type. These are assigned to types sequentially, starting with zero for the top of the lattice.

A-2.2 User Functions

(**SATISFIES?** *x type*) ⟶ *boolean*
Returns `#T` (true) if *x* satisfies *type*; returns `#F` (false) if *x* doesn't satisfy *type*; and returns the ignorance token (which can be checked for by `DUNNO?`) if *x* is undetermined for *type*.

(**IN?** *x type*) ⟶ *boolean*
Returns `#T` (true) if *x* satisfies *type* and returns `#F` (false) otherwise. This leaves out indeterminancy, converting it to `#F` (false).

(**COLLECTION-ELEMENTS** *collection-type*) ⟶ *list-of-elements*
Returns the elements of the collection type *collection-type*.

(**COLLECTION-MODIFY!** *element collection in-out)*
If *in-out* is `#F` (false), remove *element* from the mutable collection *collection*; otherwise, add *element* to *collection*. This will change the performance of `SATISFIES?`, `COLLECTION-ELEMENTS`, and (unless *in-out* is `#F` (false)) the predicate `IN?`. If *collection* is not mutable this signals an error.

(**PUT-IN-COLLECTION!** *element collection)*
Adds the object *element* to the mutable collection type *collection*. If *collection* is not mutable this signals an error.

(**TAKE-FROM-COLLECTION!** *element collection)*
Removes the object *element* to the mutable collection type *collection*. An error is signalled if *collection* is immutable or an 'empirical collection' (Section 4.4.2; Page 32).

A-2.3 The Lattice

Type descriptions are placed in a lattice of predicate subsumption; each type description stores its immediate generalizations and specializations in this lattice. While the relationship between any two types in the lattice is fixed, these stored generalizations and specializations are subject to change. In particular, if X is immediately below Z and Y is defined so as to be between X and Z, the immediate generalizations of X and the immediate specializations of Z will change to reflect the presence of Y.

The procedures described here are useful for examing the lattice. Those which simply check subsumption in the lattice are true functions; they will return the same thing regardless of additions to the lattice. (This is part of the lattice's contract.) On the other hand, the procedures which access immediate generalizations are always subject to change as the lattice is extended and filled out.

(<<? *type-a type-b*) ⟶ *below?*
Returns #T (true) if *type-a* is below *type-b* in the lattice subsumption; that is, if satisfaction of *type-a* entails satisfaction of *type-b*. The parameters *type-a* and *type-b* are coerced by ->TD before being operated upon, allowing predicates or integer identifiers to be used in place of actual type descriptions. The use of integer identifers is a useful trick for interactive use, since every type's printed representation provides the integer identifier by which it may be referred to. The procedure TD-<<? is a version of <<? which does not do type checking or attempt coercion and so is mildly faster.

(MINIMAL-TYPE-SET *list-of-types*) ⟶ *minimal-list-of-types*
Takes a list of types and returns a reduced list of types whose generalizations contain all of the original types. In particular, if any type in the list is a generalization of another type, that first type is removed since it is already included in the set by extension of generalizations.

(MAXIMAL-TYPE-SET *list-of-types*) ⟶ *minimal-list-of-types*
Takes a list of types and returns a reduced list of types whose specializations contain all of the original types. In particular, if any type in the list is a specialization of another type, that first type is removed since it is already included in the set by extension of specializations.

(GENERALIZATIONS *type*) ⟶ *list-of-types*
Takes a type and returns its immediate generalizations in the lattice. Like <<?, an attempt is made to coerce *type* if it is not a type description. As mentioned above, *list-of-types* for a given *type* is likely to change with time as the lattice is extended and filled in. The name GENZNS is an alias for GENERALIZATIONS.

(SPECIALIZATIONS *type*) ⟶ *list-of-types*
Takes a type and returns its immediate specializations in the lattice. The same coercion and caveat constraints apply as for GENERALIZATIONS. The name SPECZNS is an alias for SPECIALIZATIONS.

A-2.4 Uhhh... Indeterminancy

TYPICAL uses a special token to indicate indeterminate truth values; this token is a list stored internally to several procedures. The only access to it should be through these procedures.

(%UNDEFINED) ⟶ *ignorance-token*
Returns the token representing indeterminancy.

(DEFINED? *thing*) ⟶ *yes-or-no*
Returns #F (false) if *thing* is the indeterminancy token, #T (true) otherwise.

(**UNDEFINED?** *thing*) ⟶ *yes-or-no*

Returns `#T` (true) if *thing* is the indeterminancy token, `#F` (false) otherwise.

A-2.5 Disjointness

Disjointness is stored as an incidental property of types. Like generalization and specialization links, disjointness is only stored locally; broader disjointness is inferred by the predicate `DISJOINT?`. In particular, if any generalizations of two types are disjoint, the types are disjoint.

(**MAKE-DISJOINT!** ... *types* ...)

Declares that *types* are disjoint; i.e. that nothing satisfying one particular type in *types* satisfies any other types in *types*.

(**DISJOINT?** *type1 type2*) ⟶ *disjoint?*

Returns `#T` (true) if *type1* is disjoint from *type2*, `#F` (false)otherwise.

A-2.6 Mappings

Mappings are used in TYPICAL to define new types by constraining the image of a particular mapping to some existing type. In order to do this effectively, TYPICAL must know the domain and range of its mappings. Mappings are scheme procedures with explicit domains and ranges. Many scheme types are complicated conjunctions of other types which constrain the image of various mappings; there exist functions for extracting this information from the lattice.

(**DECLARE-MAPPING!** *procedure domain range)*

Declares the SCHEME procedure *procedure* to be a mapping from *domain* to *range*. Every image constraint over the mapping specified by *procedure* will eventually have *domain* as a generalization; there may, however, be other generalizations on the way up.

(**DETERMINE-IMAGE-CONSTRAINTS** *mapping type*) ⟶ *list-of-types*

Returns all the image constraints placed on *mapping* by *type*. This procedure climbs the lattice, collecting the constraining image for all the image constraints which constraint *mapping*, returning the corresponding list of types.

(**MAPPING-CONSTRAINT** *mapping type*) ⟶ *type-conjunction*

Returns a single type which is the conjunction of the individual constraints placed on a mapping by a type.

A-2.7 Combinators

TYPICAL's combinators are the core of its type defining abilities. Combinators may be either called directly as procedures or by another procedure like `TYPE` above.

(**SIMPLE-TYPE** *predicate generalization*) ⟶ *a-simple-type*
Returns a specialization of *generalization* which is satisfied by objects for which *predicate* returns `#T` (true); unsatisfied by objects for which *predicate* returns `#F` (false); and undetermined by objects for which *predicate* returns the ignorance token.

(**PRIMITIVE-SET-OF** *list-of-members generalization*) ⟶ *fixed-collection*
Returns a specialization of *generalization* which is only satisfied by elements in *list-of-members*.

(**QUERY-TYPE** *name-as-string generalization*) ⟶ *query-type*
Returns a specialization of *generalization* satisfied by objects which the users claims are in the class defined by *name-as-string*. For example:

```
(define crocks (query-type "a crock" symbols))
Value: CROCKS
(in? 'typical crocks)
>> Question: Is TYPICAL a crock? yes
#!TRUE
```

(**POWER-SET** *of-type*) ⟶ *power-set*
Returns a type satisfied by subtypes of *of-type*.

(**COMPLEMENT** *of-type*) ⟶ *complement*
Returns a type satisfied by objects not satisfying *type* and not satisfied by objects satisfying *of-type*. If *of-type* is undetermined for an object, the complement is also undetermined.

(`<AND>` ... *types* ...) ⟶ *conjunction*
Returns a type which is satisfied by objects satisfying every member of *types*.

(`<OR>` ... *types* ...) ⟶ *disjunction*
Returns a type which is satisfied by any object which satisfies at least one of the elements of *types*.

(**IMAGE-CONSTRAINT** *mapping type*) ⟶ *image-constraint*
Returns a type which is satisfied by objects for which *mapping* returns an object which satisfies *type*.

(**RECORD** *mapping type* ...) ⟶ *conjunction-of-image-constraints*
Returns a type satisfied by objects for which every *mapping* satisfies every corresponding *type*.

(**CROSS-PRODUCT** ... *element-constraints* ...) ⟶ *cross-product*
Returns a type which is satisfied by lists whose elements each lie in the corresponding *image-constraint.*

(**DIVIDED-TYPE** *in-test out-test beneath*) ⟶ *divided-type*
Returns a specialization of *beneath* which is satisfied by objects for which *in-test* returns `#T` (true); not satisfied by objects for which *in-test* returns `#F` (false) and *out-test* returns `#T` (true); and undetermined for everything else.

(**GENERATED-COLLECTION** *beneath*) ⟶ *generated-collection*
Returns a specialization of *beneath* all of whose elements (at any particular moment) are known.

(**COLLECTION-GENERATOR** *collection generator*) ⟶ *new-procedure*
Not strictly a type combinator, this procedure returns a copy of *generator* which adds its results to the collection *collection* which should be a generated collection.

(**DIVIDED-COLLECTION** *beneath*) ⟶ *mutable-collection*
Returns a specialization of *beneath* for which definite elements and non elements are known. Elements can be declared *in* or *out* of the resulting type by the procedure `MODIFY-COLLECTION!` or its siblings.

(**EMPIRICAL-COLLECTION** *beneath*) ⟶ *mutable-collection*
Returns a specialization of *beneath* which may be modified by the `PUT-IN-COLLECTION!` procedure and enumerated by `COLLECTION-ELEMENTS`. Any objects not explicitly added to this collection are undetermined by the type; `SATISFIES?` will never return `#F` (false) for this type.

A-2.8 Indexer Functions

This section documents the procedures provided by the actual implementation of the classifier described in Chapter 5. These functions are also used in the examples of Chapter 6.

(**MAPTYPES** *procedure object)*
Applies *procedure* to each type satisfied by *object*, in subsumption order. This means that if a type *s* is below a type *g* in TYPICAL's lattice, *procedure* will be called on *s* before *g*.

(**INDEX** *object*) ⟶ *object*
Executes daemons for each of the types which *object* satisfies. The daemons are called in subsumption order; if a type *s* is below a type *g* in the lattice, the daemons for *s* will be called before the daemons for *g*. Each daemon is called with the argument *object* and the call to **INDEX** returns *object.*

(**ADD-DAEMON!** *procedure type)*
Adds the daemon *procedure* to *type*. Whenever an object satisfying *type* is indexed beneath one of *type*'s generalizations, *procedure* will be applied to the object (except if it is particular inhibited for the object).

(**REMOVE-DAEMON!** *procedure type)*
Removes the daemon specified by *procedure* from *type*. This is not at all clever about removing the consequences produced by *procedure*'s previous actions on instances of *type*.

(**INHIBIT-DAEMON!** *daemon object ... description ...)*
Inhibits the application of *daemon* to *object*. When *object* is indexed and the daemon procedure *daemon* encountered, it will not be called. If daemons are being traced at this point, the remaining arguments to `INHIBIT-DAEMON!` (*description* above) will be passed to `PRINTOUT`.

(**DF** *object)*
Prints a description of *object* based on its location in the lattice.

(**EF** *object)*
Edits *object* based on its location in the lattice. This editor presents properties and offers commands based on the types the object satisfies.

A-2.9 Utility Procedures

(**PRINTOUT** ... *print-tokens ...)*
This is a routine for formatted printing provided in the support functions for TYPICAL. Its arguments specify a list of *printout tokens*. Each of these is printed by the SCHEME output routine `DISPLAY` (like Common Lisp's PRINC) unless it is in a special class of *execute tokens*. Execute tokens are objects which may be produced by `PRINTOUT` support procedures or bound to identifiers. For instance, the SCHEME identifier `$NL` is bound to an execute token which produces a newline, so that `PRINTOUT` would do the following:

```
(printout $NL "Foo" "Bar" $NL (+ 2 3) $NL "Bletch" $NL)
FooBar
5
Bletch
```

(**MAKE-MUTABLE**) ⟶ *mutable-procedure*
Returns a single argument procedure which maintains a table mapping objects into other objects. This mapping can be modified by the `MUTATOR` of the procedure.

(**MUTATOR** *mutable-procedure*) ⟶ *mutator-procedure*
Given a mutable procedure, returns a procedure which — given an object and a function — applies the function to the current mapping of the object to return a new value for the mapping. E.G. if `REFERENCE-COUNT` where a mutable procedure, then the following scenario could be imagined.

```
(REFERENCE-COUNT 'X)
5
((MUTATOR REFERENCE-COUNT) 'X 1+)
5
(REFERENCE-COUNT 'X)
6
```

(**MODIFIER** *mutable-procedure*) ⟶ *modifier-procedure*
Given a mutable procedure, returns a procedure which — given an object and a value — modifies *mutable-procedure* so that the mapping of the object will be the value. As in the example above, we could use this on `REFERENCE-COUNT` as:

```
((MODIFIER REFERENCE-COUNT) 'X 0)
```

to clear the reference count for the symbol `X`.

This empty page was substituted for a blank page in the original document.

Chapter A-3
Getting TYPICAL

You can get copies of TYPICAL to experiment with or use from a variety of locations. Of course, no warranties or guarantees are expressed or implied by such availability.

TYPICAL is implemented in SCHEME, using an extended superset of the standard SCHEME defined in [RC86]. Most of TYPICAL's development was done in `C-SCHEME`, an MIT SCHEME implemention in `C` which runs under a variety of operating systems. The easiest way to get a copy of TYPICAL is to get (or discover that you already have access to) the MIT C-Scheme Release 5.3 or later. This can be FTP'd (in Unix `TAR` format) from the Internet host "MIT-PREP" as the file "`/scheme/dist.tar`". This distribution gives you all of C-Scheme along with the subdirectory "`libs/kwh`" (for the author's initials) which contains the sources to TYPICAL. If you don't have Internet access, you can get a tape (for a tape hassling cost of $200.00) of CScheme from:

Scheme Distribution
c/o Professor Harold Abelson
Massachussets Institute of Technology
545 Technology Square
Cambridge, Massachussetts 02139

In addition, C-Scheme (and TYPICAL) are included in the standard release of 'GNU Emacs' from the Free Software Foundation. If you have a post-December '87 release of GNU Emacs you may already have a copy of TYPICAL; if not you can order a GNU Emacs release (for a $150.00 tape handling charge) from:

Free Software Foundation
1000 Massachussetts Avenue
Cambridge, MA 02138

Finally, if you don't have or want either of these, you can FTP a copy of TYPICAL from the Internet hosts "MIT-REAGAN" (in the directory "`>KWH>Distribution>`") or "MIT-PREP" (in the directory "`/u/kwh/distribution/`"). If you want to run TYPICAL in Common Lisp [Ste84], you probably want to get a copy of Jonathan Rees's PseudoSCHEME (in Common Lisp); you can get this from the Internet host MIT-MC in the file "`MC:JAR;PSEUDO >`."

A-3.1 The Files

Each of the sources above will give you a set of three or four subdirectories: a directory of language-dependent SCHEME extensions, a directory of language-independent SCHEME extensions, the actual code of TYPICAL (including the indexer described in Chapter 5), and (maybe) a snapshot of **Cyrano**'s current development. While you are welcome to try running **Cyrano**, remember that it is a snapshot of a research program in development.

A-3.1.1 Scheme (R^3S) extensions

TYPICAL is implemented in a superset of the standard Scheme described in [RC86]. The extensions used by TYPICAL are:

1. A family of operations on bit strings which support an abstraction of infinite length bit strings (with trailing zeros) with 'bit-setting' and bitwise-logical operations on them.
2. A syntactic `definline` which encourages the inline coding of various definitions.
3. A fluid binding special form `FLUID-LET` which provides for dynamic binding of variables.
4. A record structure definition macro `DEFINE-STRUCTURE` which defines a composite objects with accessor and modifier functions and provides the structure with a special printing format.
5. Timing functions for returning either running time in 100ths of seconds (`SYSTIME`) or time of day as a list (`HOURS-MINUTES-SECONDS`).
6. A collection of *lookup* functions for creating, using, and modifying lookup tables.

The subdirectory "`plus`" contains a collection of files for extending various [RC86] standard Schemes to support this superset:

1. "`plus.scm`" contains the extensions for MIT's C-Scheme.
2. "`plus.t`" contains the extensions for `T`, the Yale dialect of Scheme.
3. "`plus.l`" contains the extensions for Jonathan Rees's PseudoScheme implemention of Scheme in Common Lisp.

If you want to run TYPICAL in another Scheme implementation, you can look at these files to implement the appropriate extensions for your dialect. The C-Scheme implementa-

tion also requires some extensions written in C. The file "`bitops.c`" contains C-extensions to C-Scheme that implement the bit string primitive operations used by `TYPICAL`.[15]

A-3.1.2 Scheme utilities

The subdirectory "`utilities`" contains a collection of Scheme utilities used by TYPICAL. These utilities use the above-described extensions to the [RC86] standard. There are seven files:

- The file `mapfcns.scm` defines a collection of procedures for operating on or over common data structures, including `MAPTREE`, `MERGE`, `COLLECT`, `UNION`, etc.
- The file `mutable.scm` provides a facility for generating *mutable procedures* which can be used as lookup tables for various properties.
- The file `tuple.scm` implements the 'tuple' data type; the tuple is a sort of 'hash-consed' list. Two tuples with `EQ?` elements are `EQ?`. This file also implements a list canonicalization routine used to 'memoize' procedure calls.
- The file `printout.scm` contains a formatted printing facility inspired by InterLisp's PRINTOUT. `PRINTOUT` is an extensible expression oriented formatting command providing much of the functionality of Common LISP's `FORMAT` in a cleaner fashion.
- The file `message.scm` implements a special version of `PRINTOUT` which is used for describing program events to a user.
- The file `switches.scm` provides a facility for software swtiches which can be set or reset by the user.
- The file `engine.scm` implements a tasking facility for TYPICAL which allows the specification of procedures which divide their work over multiple invocations.

Some of these utilities are used directly by TYPICAL while others are used only by **Cyrano**.

A-3.1.3 TYPICAL Sources

The subdirectory `typical` contains the sources for TYPICAL and a few utilities (including the indexer described in Chapter 5) implemented with TYPICAL.

- The file `kernel.scm` contains the core of TYPICAL's implementation, specifiying the 'type description' record structure and the procedure `TYPE-GENERATOR` for defining new type combinators. (Described in Section 3.2 (Page 18).)
- The file `synthetic.scm` implements TYPICAL's synthetic combinators. This also implements the procedures for modifying and enumerator collection types. (Described in Section 4.4 (Page 30).)
- The file `direct.scm` implements TYPICAL's direct analytic types. (Described in Section 4.1 (Page 23).)

[15] In C-Scheme, you can do a "`make kwhscheme`" in the Scheme microcode directory to get a version of SCHEME (called "`kwhscheme`") with these primitives built in.

- The file `indirect.scm` implements TYPICAL's indirect analytic types. (Described in Section 4.2 (Page 27).)
- The file `inductive.scm` implements TYPICAL's inductive definition types. (Described in Section 3.2 (Page 18).)
- The file `datatypes.scm` specifies types and mappings corresponding to the primitive scheme data types.
- The file `metatypes.scm` specifies types of types and declares — as valid mappings — various type related functions.
- The file `test-suite.scm` contains the test suite for TYPICAL described in Section 2.2. (Described in Section 2.2 (Page 10).)
- The file `metafns.scm` defines a simple higher level language which does a degree of automatic type inference (e.g. about composed or mapped functions).
- The file `maptypes.scm` implements the `MAPTYPES` procedure described in Chapter 5 (Section 5.2.1 (Page 43)).
- The file `index.scm` contains the indexer described in Chapter 5.
- The file `df.scm` implements a "DESCRIBE" command (`df`) which uses the lattice of types to determine what properties to describe.
- The file `props.scm` specifies describable properties of SCHEME and TYPICAL objects; these are used by `DF` to determine how particular objects should be described.
- The file `ef.scm` implements a data structure inspector which — like the description command above — determines properties and operations based on the indexing of objects in TYPICAL's lattice.
- The file `comands.scm` specifies commands applicable to various SCHEME and TYPICAL objects; these are accessible in `EF` as operations on objects being edited.
- The file `scode.scm` defines an extension fo `EF` in C-Scheme which allows the editing of internal variables of S-Code procedures.

Chapter A-4
References

[AS85] Harold Abelson and Gerald J. Sussman. *The Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[BS85] Ronald J. Brachman and James G. Schmolze. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, 13, 1985.

[BW77] Daniel. Bobrow and Terry. Winograd. An Overview of KRL: A Knowledge Representation Language. *Cognitive Science*, 1(3), 1977.

[Car83] Lucas Cardelli. ML Under Unix. *Polymorphism*, I(3), 1983.

[Cha83] David Chapman. *Naive Problem Solving and Naive Mathematics.* Working Paper 249, MIT Artificial Intelligence Laboratory, 1983.

[Cha86] David Chapman. *Cognitive Cliches.* Working Paper 286, MIT Artificial Intelligence Laboratory, 1986.

[Gol76] Ira Goldstein. *FRL: A Frame Representation Language.* Memo 333, MIT Artificial Intelligence Laboratory, 1976.

[Gre80] Russel Greiner. *RLL-1: A Representation Langauge Language.* Working Paper 80-9, Stanford Heuristic Programming Project, October 1980.

[Haa86] Ken Haase. *ARLO: Another Representation Language Offer.* Technical Report 901, Artificial Intelligence Laboratory, MIT, 1986.

[Haa87] Ken Haase. *Why Representation Language Languages are No Good.* AI Memo 943, Artificial Intelligence Laboratory, MIT, 1987. (In preparation).

[LAB*79] Barbara Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, B. Scheifler, and A. Snyder. *CLU Reference Manual.* Techical Report 225, MIT Laboratory for Computer Science, 1979.

[Len83] Douglas B. Lenat. Eurisko: A Program which Learns New Heuristics and Domain Concepts. *Artificial Intelligence*, 21, 1983.

[McA87] D. McAllester. *ONTIC: A Mathematical Representation Language.* Technical Report 979, Artificial Intelligence Laboratory, MIT, 1987.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *JCSS*, 17(3), December 1978.

[Mil83] R. Milner. A Proposal for Standard ML. *Polymorphism*, I(3), 1983.

[Min86] Marvin Minsky. *The Society of Mind.* Simon and Schuster, 1986.

[Nau61] P. Naur. Report on the Algorithmic Language Algol 60. *Communications of the ACM*, 3, 1961.

[RC86] Jonathan Rees and William Clinger. *The Revised^3 Report on the Algorithmic Language Scheme.* Memo 848a, Artificial Intelligence Laboratory, MIT, 1986.

[RS76] Charles. Rich and Howard E. Shrobe. *Initial Report On a LISP Programmers Apprentice.* Technical Report 354, Artificial Intelligence Laboratory, MIT, 1976.

[Ste79] M. Stefik. An Examination of a Frame Structured Representation System. In *Proceedings of the Sixth IJCAI*, IJCAI, 1979.

[Ste84] Guy L. Steele. *Common LISP: The Language.* Digital Press, 1984.

[Wir71] N. Wirth. The Programming Language Pascal. *Acta Informatica*, 1, 1971.

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 9/28/95

Report # AI-TR-988

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 111 (118-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAPS	(1-111) UNN#TITLE, UNN#BLK, i, UNN#BLK, UNN#DED, CAT, UNN#BLK, ii-vi, UNN#BLK, UNN#BLK, 1-3, UNN#BLK, 5-19, NO PAGE #20, 21-35, UNN#BLK, 39-47, UNN#BLK, 49-83, UNN#BLK, 85-93, UNN# BLK, 95-100 (112-118) SCANCONTROL, COVER, DOD (2), TRGTS (3)

Scanning Agent Signoff:
Date Received: 9/28/95 Date Scanned: 10/5/95 Date Returned: 10/5/95

Scanning Agent Signature: Michael W. Cook

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AI-TR 988
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): TYPICAL: A Knowledge Representation System (based on Type Specification) for Automated Discovery and Inference

5. TYPE OF REPORT & PERIOD COVERED: A.I. Technical Report

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): Kenneth W. Haase Jr.

8. CONTRACT OR GRANT NUMBER(s): N00014-85-K-0124

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Artificial Inteligence Laboratory
545 Technology Square
Cambridge, MA 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS:
Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

12. REPORT DATE: August 1987

13. NUMBER OF PAGES: 110

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office):
Office of Naval Research
Information Systems
Arlington, VA 22217

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report):

Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):

18. SUPPLEMENTARY NOTES:

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number):

Knowledge Representation, Discovery, Type Inference, Subsumption

20. ABSTRACT (Continue on reverse side if necessary and identify by block number):

TYPICAL is a package for describing and making automatic inferences about a broad class of SCHEME predicate functions. These functions, called types following popular usage, delineate classes of primitive SCHEME objects, composite data structures and abstract descriptions. TYPICAL types are generated by an extensible combinator language from either existing types or primitive terminals. These generated types are located in a lattice of predicate subsumption which captures necessary

Block 20 continued;

entailment between types; if satisfaction of one type necessarily entails satisfaction of another, the first type is below the second in the lattice. The inferences made by TYPICAL are relations in this lattice of subsumption; when a type is defined, TYPICAL computes the position of the new definition in the lattice and establishes it there. This information is then accessible to both later inferences and other programmes (reasoning systems, code analysers etc.) which may need the information for their own purposes. TYPICAL was developed as a representation language for the discovery programme CYRANO; particular examples are given of TYPICAL'S application in the CYRANO programme.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**